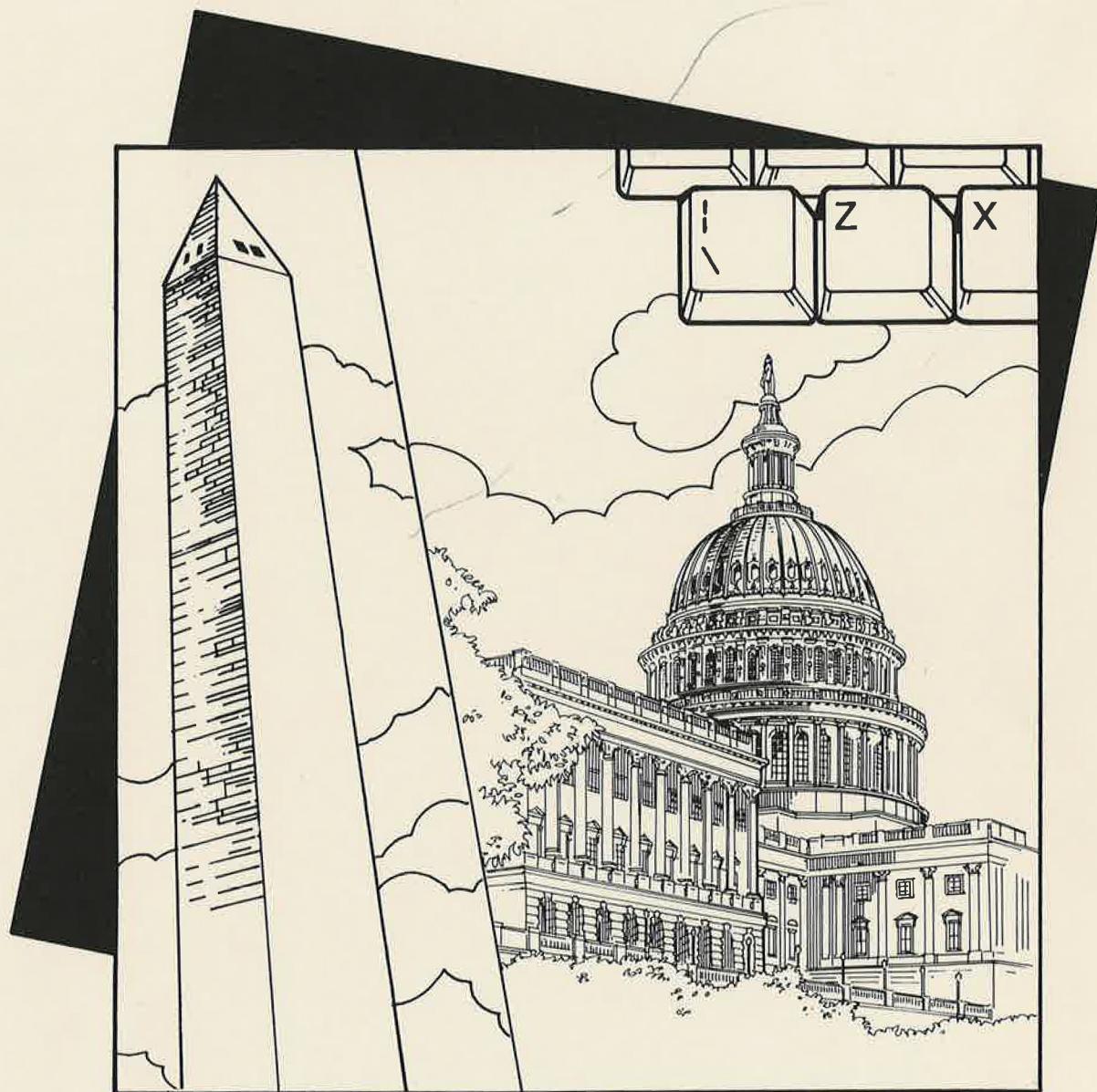




The Professional and Technical UNIX® Association  
*UNIX® is a registered Trademark of AT&T*



**Winter 1987 USENIX Technical Conference  
Washington, D.C.**

**CONFERENCE PROCEEDINGS**

Winter 1987 USENIX Association Conference Proceedings, Washington, D.C.



USENIX Association

Winter Conference Proceedings

Washington, D.C. 1987

January 21-23, 1987

Washington, District of Columbia, USA

For additional copies of these proceedings, write:

USENIX Association

P.O. Box 7

El Cerrito, CA 94530 USA

Price \$20.00 plus \$25.00 for overseas airmail

© Copyright 1987 by The USENIX Association

All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain  
with the author or the author's employer.

UNIX is a registered trademark of AT&T

Other trademarks noted in the text.

# ACKNOWLEDGEMENTS

**Sponsored by:**

The USENIX Association  
P.O. Box 7  
El Cerrito, CA 94350

## WHAT IT IS TO BE UNIX: *A few simple notions and where they lead*

**Program Chairperson:**

David Tilbrook, *Quantime, Ltd.*

**Program Assistant Chairperson:**

Deborah Scherrer, *mt Xinu, Inc.*

**Program Committee:**

Neil Groundwater, *Sun Microsystems, Inc.*

Nigel Martin, *The Instruction Set*

Jim McKie, *Bell Communications Research*

Kirk McKusick, *University of California, Berkeley*

Norman Wilson, *AT&T Bell Laboratories*

## UNIX PERFORMANCE

**Program Chairperson:**

Herb Schwetman, *MCC*

**Program Committee:**

Bob Brown, *NASA-RIACS*

John Chambers, *MCC*

John Quarterman, *Consultant*

## UNIX-BASED DATA MANAGEMENT SYSTEMS

**Program Chairperson:**

Paula Hawthorn, *Britton Lee, Inc.*

**Program Committee:**

David DeWitt, *University of Wisconsin*

Judith Obarr, *Britton Lee, Inc.*

Lou Katz, *Metron Computerware, Ltd.*

**TUTORIAL COORDINATOR:**

John Donnelly

**USENIX MEETING PLANNER:**

Judith F. DesHarnais

**CONFERENCE HOST:**

Rick Adams, *Center for Seismic Studies*

**PROCEEDINGS PREPARATION:**

Donnalyn Frey, *Mazim Technologies, Inc.*

Rick Adams, *Center for Seismic Studies*

# Table of Contents

**What it is to be UNIX: A few simple notions and where they lead**

Wednesday, January 21, 1987

<b>9:00 A.M.</b>	<b>Welcome and Overview of Session</b> David Tilbrook, Deborah Scherrer	
<b>9:20 A.M.</b>	<b>UNIX: Leverage – Past, Present, Future</b> John R. Mashey, <i>MIPS Computer Systems</i>	1
<b>10:30 A.M.</b>	<b>UNIX: The Data Forms</b> Eric Allman, <i>Britton Lee, Inc.</i>	9
<b>11:10 A.M.</b>	<b>UNIX: The Language Forms</b> Stephen C. Johnson, <i>Dana Computer, Inc.</i>	16
<b>11:50 A.M.</b>	<b>UNIX and Networking: A Separate Peace</b> John R. Mullen, <i>Communication Machinery Corp.</i>	21
<b>2:00 P.M.</b>	<b>UNIX: The Cult</b> Peter Collinson, <i>University of Kent, Canterbury</i>	22
<b>2:40 P.M.</b>	<b>UNIX: A Dialectic</b> Dennis M. Ritchie, <i>AT&amp;T Bell Laboratories</i>	29
<b>3:20 P.M.</b>	<b>UNIX: The World View</b> Mike O'Dell, <i>Maxim Technologies</i>	35
<b>4:30 P.M.</b>	<b>Question-and-Answer Panel</b> Speakers and some members of Program Committee	

# UNIX Performance

Thursday, January 22, 1987

8:30 A.M.	<b>Keynote Speaker</b> <b>Managing the Development of Performance-Constrained UNIX-based Software on Microcomputers</b> Lawrence B. Perkins, <i>Martin Marietta Corporation</i>	46
9:00 A.M.	<b>Session 1</b> <b>Experimental Sensitivity Analysis of Performance in a UNIX System</b> Lindsey E. Stephens and Lawrence W. Dowdy, <i>Vanderbilt University</i>	60
	<b>An Experimental Assessment of Resource Queue Lengths as Load Indices</b> Songnian Zhou, CSRG, <i>University of California, Berkeley</i>	73
	<b>A Parallel Programming Process Model</b> Bob Beck and Dave Olien, <i>Sequent Computer Systems</i>	83
11:00 A.M.	<b>Session 2</b> <b>A Prototype Capacity Planning and Configuration Modeling Tool for UNIX Systems</b> G. Ramamurthy and Y.T. Wang, <i>AT&amp;T Bell Laboratories</i> Hank Nichols and Mike Andrews, <i>AT&amp;T Information Systems</i>	103
	<b>A Knowledge-based System for Performance Tuning of the UNIX Operating System</b> Behrokh Samadi, <i>AT&amp;T Bell Laboratories</i>	110
	<b>Software Performance Analysis Using Call Graphs and Workstation Graphics</b> David B. Leblang, <i>Apollo Computer</i>	124
2:00 P.M.	<b>Session 3</b> <b>ILMON: A UNIX Network Monitoring Facility</b> Lewis Barnett and Michael K. Mallor, <i>University of Texas, Austin</i>	133
	<b>Virtual Disks: A New Approach to Disk Configuration</b> Thomas Van Baak, <i>Pyramid Technology Corporation</i>	145
	<b>Disk Response Time Measurements</b> Thomas D. Johnson, Johnathan M. Smith and Eric S. Wilson, <i>Bell Communications Research</i>	147
4:00 P.M.	<b>Session 4</b> <b>Tuning UNIX Lex or It's NOT True What They Say About Lex</b> Van Jacobson, <i>Lawrence Berkeley Laboratory</i>	163
	<b>Methodology and Results of Performance Measurements for a New UNIX Scheduler</b> Jeffrey H. Straathof, Ashok K. Thareja, and Ashok Agrawala, <i>University of Maryland</i>	165
	<b>News Need Not Be Slow</b> Geoff Collyer and Henry Spencer, <i>University of Toronto</i>	181

# UNIX-based Data Management Systems

Friday, January 23, 1987

9:00 A.M.	<b>Introduction</b> Paula Hawthorn, <i>Britton Lee</i>	
9:10 A.M.	<b>Keynote</b> <b>Database Management Systems: Past, Present and Future</b> Philip Bernstein, <i>Wang Institute of Graduate Studies</i>	—
10:15 A.M.	<b>Distributed Database Systems: Why Distributed File Systems Are Not Enough</b> Robert McCord, <i>Relational Technology, Inc.</i>	—
10:45 A.M.	<b>Database Machines: Why Fast File Systems Are Not Enough</b> Robert Taylor, <i>Britton Lee, Inc.</i>	—
11:15 A.M.	<b>Round-Table Discussion of Issues Raised in Morning Talks</b> Morning Session Participants	
1:00 P.M.	<b>Data Management: A Full-Text Information Retrieval Perspective</b> Ken Leese, <i>Fulcrum Technology</i>	191
1:45 P.M.	<b>UNIX in Health Care: Medical Laboratories – A Case Study</b> Sheldon Hamburger, <i>Management Systems Engineering, Inc.</i>	192
3:00 P.M.	<b>RDBMS Features and Data Integrity Issues in an Army Budget Database System</b> Patricia J. Ton, <i>ITT Research Institute</i>	196
3:45 P.M.	<b>Real World UNIX DBMS Applications: Experiences and Observations</b> Stephen Tolchin, Eric Bergan and Marvin Schneider, <i>The Johns Hopkins Hospital</i>	210
4:30 P.M.	<b>Round-Table Discussion of Issues Raised in Afternoon Talks</b> Afternoon Session Participants	

## AUTHOR INDEX

Agrawala, Ashok	165
Allman, Eric	9
Andrews, Mike	103
Baak, Thomas Van	145
Barnett, Lewis	133
Beck, Bob	83
Bergan, Eric	210
Collinson, Peter	22
Collyer, Geoff	181
Dowdy, Lawrence W.	60
Hamburger, Sheldon	192
Jacobson, Van	163
Johnson, Stephen C.	16
Johnson, Thomas D.	147
Leblang, David B.	124
Leese, Ken	191
Mallor, Michael K.	133
Mashey, John R.	1
Mullen, John R.	21
Nichols, Hank	103
O'Dell, Mike	35
Olien, Dave	83
Perkins, Lawrence B.	46
Ramamurthy, G.	103
Ritchie, Dennis M.	29
Samadi, Behrokh	110
Schneider, Marvin	210
Smith, Johnathan M.	147
Spencer, Henry	181
Stephens, Lindsey E.	60
Straathof, Jeffrey H.	165
Thareja, Ashok K.	165
Tolchin, Stephen	210
Ton, Patricia J.	196
Wang, Y.T.	103
Wilson, Eric S.	147
Zhou, Songnian	73



## UNIX: Leverage — Past, Present, Future

*John R. Mashey*

MIPS Computer Systems

“Give me a place to stand and a lever long enough, and I’ll move the world.”

### 1. Introduction

The whole idea of software tools is closely associated with the UNIX® system. Much has been written on the theory and practice of UNIX tool-building and tool-using. In fact, we’ve come to take this for granted so much that it’s almost a cliché. Using the analogy of the simplest tool of all, the lever, this paper offers another perspective on the sources of leverage within UNIX, past, present, and future.

According to the encyclopedia, a lever is a simple tool used to amplify muscular strength. UNIX is a lever for the intellect. This paper first reviews the ‘place to stand’, i.e., the typical UNIX computing environments of the last few years. Then, it describes several different dimensions of leverage, to focus attention on the different ways that UNIX does (or does not) provide leverage. Thus, we can see whether we’re able to move worlds ... or just big rocks.

Finally, this paper also tries to set the stage for the later papers in this session: expect to see forward references of the form [author’s name].

### 2. A Place to Stand

I’ve previously used the following model of UNIX history:

- Roots (Pre-1969)
- Birth (1969-1973)
- Childhood (1973-1977)
- Adolescence (1977-1981)
- Maturity (1981-)

I’d now modify this to include:

- Maturity (1981-1985)
- Second Adolescence (1985-1989), or Here We Go Again

Of course, this is an over-simplification, as anyone would know who’s ever seen UNIX version history charts, even partial ones. However, it still seems to offer a useful model of events. I don’t know why the eras all ended up 4 years long; perhaps there is a natural periodicity to such things. Important events (new UNIX versions or major technology changes) tended to happen near the end of each period, then spread rapidly. This lets us choose the year just after each boundary (1974, 1978, etc) as an interesting sample point, although the particular choice needs explanation.

Let us look at the process by which new technology spreads. In step 1, some new technology is developed in some research lab or by some other small group of people. For a few years, it may be that nothing is even published about it. In step 2, other groups (usually called ‘early adopters’ in marketing parlance) begin applying the new technology. This may take years, during which the originators and early adopters run about trying to sell the ideas

to others, and are somewhat successful, but are often ignored or classed as lunatics. In step 3, exponential growth takes over, and many more people get on board. By step 4, even the most conservative people are finding uses for it, and it becomes an expected part of the environment. The sample years chosen were ones where some technology was in middle-to-late step 2. This is an interesting phase, in that new technology is strong enough that you know it is successful (unlike things that never make it past step 1), but it can be expected to grow much further.

This model leans more towards the programming and technical uses of UNIX, and also reflects my own biases and experiences.

## 2.1. 1974

UNIX users were mostly programmers working on UNIX-based projects, mostly in small, tightly related groups, using the recently-available UNIX V5. Most people were using, or moving to the DEC® PDP® 11/45, about .5 Mips, with a maximum of 248KBytes of memory. A typical configuration cost about \$250K. With a minimum (typical) maximum user load of 1(16)24 simultaneous users, each user's average share of CPU and memory was .50(.03).02 Mips, and 248KB(25KB)10KB of memory. People used 300-Baud hardcopy terminals. Support of 24 simultaneous users was accomplished by heroic system administration coupled with the patience of users desperate to accomplish at least a little work. Nevertheless, such systems were both useful and cost-effective in comparison with mainframe time-sharing, and hence were attracting interest. Addition of space-consuming features to any important program was viewed with extreme suspicion. Acquisition of a new 'release' of software was accomplished by driving to Murray Hill to see what new things could be found on the research machine.

## 2.2. 1978

Many more users were programmers and others that might be using UNIX as a general utility, or to support non-UNIX programming projects. UNIX was UNIX V6, or the USG or PWB versions thereof. People most often used a PDP 11/70 (.8Mips, 1MB memory), with 1(32)48 simultaneous users. This gave each user .80(.03).02 Mips and 1024KB(32KB)21KB. Although computing power had not increased tremendously, the larger memory was quite helpful, since programs shared space efficiently, and both kernel and user processes remained squeezed into 64KB text plus 64KB data. Adding features had certainly become thinkable. People continued to use 300-Baud hardcopy terminals, but more terminals were running at 1200-Baud, and 24X80 CRTs were becoming widespread, sometimes running at higher speeds. A useful system could still be bought for about \$250,000, a cost that happened to be low enough that a Bell Labs Director could sign for it with minimal bureaucracy. The result was a proliferation of systems bought by organizations voting with their feet. At least somewhat in response to this trend, PDP 11/70s had begun to appear inside most Bell Labs computer centers.

## 2.3. 1982

The user community now included many non-programmers, and they most likely used a 4MB (or maybe 8MB) VAX 11/780, so that 1(32)48 people received 1.0(.03).02 Mips and 4096KB (130KB) 90KB apiece. However, although they *still* hadn't gained much CPU power, many more were using CRTs at speeds high enough to allow routine use of screen editors, and once again, hardware design helped software by letting us throw more memory at the problem. UNIX variants abounded, including 4.1BSD, System III, USG 4.0 (inside BTL), and various V7 offshoots. Many people inside Bell Labs had learned portability lessons from painful experiences of porting PDP-11 code to other machines. UNIX kernels and user processes were permitted to grow beyond 64KB text plus 64KB data, thus removing the major technical barrier to creeping featurism. People were busy porting UNIX V7 to microprocessors, and the IBM PC had Happened, although most true UNIXers looked down

upon the primitive facilities available thereon. Local networks were beginning to be used, albeit haphazardly, and people were looking at bitmapped displays for UNIX.

#### 2.4. 1986 (NOW)

Unlike earlier years, when it was easy to show a canonical machine environment for UNIX, it is now much more difficult to paint a dominant mode of usage, even in the technical area alone. People may still share a mainframe, mini-super, or large supermini. For example, 1(32)80 might share a 20MB, 4.2 Mips VAX 8600, so that each receives 4.2(.13).05 Mips, and 20MB(.7MB).25MB. A super-micro may be shared by a smaller number of people, so that a 2 Mips CPU with 4MB memory is shared by 1(8)16 people, giving each 2(.25).12 Mips, and 4MB(.5MB).25MB of memory. Finally, single-user UNIX machines are much more common, ranging from PCs to technical workstations, which give the individual user 1-2 Mips and 4MB or more of memory. Much computing is now done in heterogeneous networks. Options and features have proliferated, but the improved user interfaces offered by higher-bandwidth displays help mitigate this effect upon the user. Companies tolerating, using, or owing their existence to UNIX have also proliferated. UNIX versions have consolidated around System V Release 2 or 4.2BSD, with SVR3 and 4.3BSD starting to appear.

#### 2.5. 1990

We can safely assume that the computing environment will have diversified even more than it has already. Some groups of people will always seek minimal-cost solutions, and will therefore either use low-cost PCs with file servers or divide a larger computer. Presumably applications will grow, so that most people will get (and need!) at least .5Mips and .5MB apiece. Anyone who needs more power will be able to buy a 10-20 Mips, 16-64MB single-user workstation for entry costs of less than \$1,000 / Mips, and such things will be in widespread use, meaning that they better have started to appear by 1988. Small servers (i.e., still not necessarily in the computer room) will offer 30-50 Mips and 64MB-512MB, at costs of \$1,000-\$2,000 / Mips. High-end single-user workstations should offer similar performance. Mini-supers will have gotten more super.

Almost anything that is not currently networked will be, ranging from inexpensive low-speed nets for the office to high-speed fiber-optic nets among bigger workstations and servers. Almost every new computer architecture will run UNIX. However, kernel underpinnings may have changed to ones like V (Stanford) or Mach (CMU) that are more directly aimed at distributed systems. There will still be no standard UNIX, although many will have incorporated SVID, X/OPEN, or POSIX. As always, there will be a horde of UNIX variants that differ in those areas nearer the state of the art. Thus, it is likely that there *still* will exist several competing standards for graphics, and especially for windowing, but the existence of several standards will be a great improvement over the current situation, in which there are too many, or none, depending on your viewpoint. Networking should improve in a similar fashion. There will probably be hordes of competing representations for complex documents, image, and voice.

Note that the speed and cost predictions are *conservative* ones, based on reasonable technology trend analyses. However, applications will have grown to consume this power, just the way they jumped from 11/70s to VAXen, and immediately gobbled everything. It is safe to predict that some people will expect much more.

Now, let's go back and look at different areas in which UNIX has given, does give, will give, or ought to give us leverage. The later articles go into more depth on some of the same topics. In each section I offer a summary, setting 1974 UNIX to <5>, then rating the rest of computing and later UNIX systems on that scale. Bigger numbers are better; the scale is arbitrary; all opinions are my own; they're mainly meant to stir discussion on where we might go.

### 3. Individual Programming

"Give ME a Lever..."

The earliest place to look in UNIX for leverage is for the individual programmer, i.e., in aspects that improve an individual's productivity.

#### 3.1. Editors, Interfaces, and Data [Allman, O'Dell]

Here we cover the fundamental leverage area of the energy required to make the machine do something, specifically in terms of the immediate human interface [editors, command languages] and file system.

		UNIX	Rest
	1974	3-5	0-6
Leverage	1978	5-7	0-8
summary	1982	5-8	0-10
	1986	8-12	0-15
	1990	8-20	0-20

- Even early UNIX offered the programmer working leverage by simply suppressing unwanted details of disk allocation and access, and by offering simple, inexpensive, and relatively powerful human interfaces.

- In 1974, the typical UNIX programmer used a 30-character-per-second hardcopy terminal, unendurable by current standards. However, 300 baud, *ed*, and *sh* were often thought wonderful <5> when compared with the alternatives. Those who've never had the experience of allocating tracks of disk space via IBM JCL cards <0> might try this once to appreciate just what a wonderful improvement UNIX was at that point. Many programmers still submitted decks of punched cards for batch processing, or used expensive, restricted line editors that lacked even regular expression pattern matching. In fact, many were pleased to be able to edit such decks as UNIX files and run them via Remote Job Entry <3>.

- By 1982, most people expected to use CRT terminals running at 1200 baud, Screen editors, such as *vi* or *emacs* became widely available <8>, and remain so today. Shells also improved, offering (early) more programmability and (later) more powerful terminal interfaces.

Although the above mode of interaction remains dominant <8>, use of bitmap displays as terminals or workstations has now become more common <10-12>, with noticeable improvements starting around 1983-1984. Such use clearly changes the potential modes of interaction to include much heavier use of display bandwidth, pointing devices, and graphics.

UNIX has certainly been a better host to advanced interfaces than have many older operating systems: many technical workstations are based on UNIX. On the other hand, UNIX is still plagued by a lack of standards for graphics window-management. Finally, compared to workers in Computer-Aided Design, software engineers are woefully underserved.

To summarize, for many years UNIX had one of the most convenient interfaces, when compared with mainframe and other minicomputer systems. For a while, others have surpassed it in some areas by running on personal computers too small for UNIX. As microprocessor system costs have dropped, UNIX has become available on single-user workstations, and is starting to catch up in immediate interfaces.

- Hopefully, by 1990, it will have interfaces as good as the best of the rest, and perhaps advantages over PC operating systems that have been extended beyond their natural design points.

### 3.2. Text Processing [Allman, Johnson]

"Why can't *troff* be more like my Mac, and vice-versa."

This has long been a strength of UNIX, but has, unfortunately, not progressed as fast as one might like. This area is quite important to programmers and most other users. In the early life of the Programmer's Workbench, it was often noted that people justified their usage by predicting programming productivity, but what they really did was documentation:

		UNIX	Rest
	1974	5	0-4
Leverage	1978	8	0-6
summary	1982	8-10	2-12
	1986	8-12	2-20
	1990	8-30	2-30

- In 1974, what most people got was raw *nroff*<5>, although a few lucky groups had phototypesetters and access to *troff*.
- By 1978, many people used *tbl*, *eqn*, and powerful formatting macro packages <8>. Larger typesetters became widely available through computer centers. In general, the UNIX toolkit was as good, and usually better, than other widely-available systems. Some new tools have been added since then <10>.
- By 1986, inexpensive laser printers are widely available. This area has supported continual engineering improvements, but has seen few real breakthroughs. For example, *troff* externals have changed little in 10 years, and several of the still-popular macro packages (—MS, —MM, for example) were written 10-12 years ago. Ease-of-use of *troff* and related programs is nowhere near that of some of the current desktop publishing systems <20>, although *troff* and its friends can still do some things the others cannot. The others are catching up fast. At least UNIX is also host to systems like Interleaf® or Frame®. An ideal system <30> would combine the convenience and ease of use of these with the expressive power and control of the *troff* complex. Unfortunately, the ideal is nowhere near in heavy use yet, and it probably awaits the more powerful workstations alluded to above, i.e., 10 Mips on the desktop.
- I do have hopes that the desktop publishing system I've always wanted will be available by 1990 on UNIX if only because more systems are being written in C.

### 3.3. Programming Languages and Level of Work [Johnson]

This is intended to measure the amount of work it takes to write, debug, and maintain new programs. Since there exist many problem domains and languages to handle them, this section is of necessity an over-simplification focused on systems programming and related activities.

		UNIX	Rest
	1974	2-5	0-4
Leverage	1978	5-10	0-12
summary	1982	7-12	0-15
	1986	8-15	0-20
	1990	8-30	0-25

- In 1974, being able to use C instead of assembler for systems programming was wonderful <5>.
- By 1978, shells had long since become programmable, other useful tools, like *make*, *awk*, and SCCS were in use <10>.
- During 1977-1981, I remember a great deal of experimentation and creation of special-purpose languages, but somehow, it didn't seem like there was that much clearly-observable widespread progress in this area <12>.

- By 1986, many more special-purpose languages exist, including the application generators and 4GLs that now routinely accompany database systems. Also, object-oriented ideas have now infiltrated UNIX, by way of C++, or Objective-C, for example <15>. However, many people still write in the same C that existed in 1978.

- In 1990, one can safely assume that raw C (in the form of ANSI standard C) will still be popular, although languages like C++ should be in heavy use. I'd hope that we'll be able to move to more highly-leveraged languages (SMALLTALK®, LISP, PROLOG), constraint-based systems (like vanWyk's *ideal*), better debuggers, and other tools that burn Mips to provide expressiveness <25>. The soon-to-occur jump in performance and cost/performance should make this possible.

#### 4. Group Programming

"Give US Some Levers...and Let Us Move Rocks Together"

These attributes measure programming as a group activity. Note that minicomputer-based operating systems have often done better than either mainframes or personal computers in this area.

##### 4.1. Shared Data [Allman, Mullen]

This attribute measures the ease of sharing data.

		UNIX	Rest
	1974	5	0-6
Leverage	1978	7	0-8
summary	1982	10-12	0-12
	1986	10-15	0-16
	1990	10-25	5-25

- In 1974, UNIX was much stronger than most systems in offering convenient sharing of files <5>.

- By 1978, some modest improvements had been made in areas of protection and ease-of-use when multiple groups shared machines <7>.

- By 1982, various groups had implemented networked filesystems of various ilks; that they existed was good; that they remained of various ilks was bad <10-12>. Various vendors (Apollo, Prime, DEC, for example) had implemented some good homogeneous networked filesystems.

- By 1986, there exist several vendor-supported proprietary UNIX-based network filesystems. AT&T's RFS has been released, and Sun's NFS is widely used <10-15>.

- By 1990, most UNIX systems will support some network file system, probably either NFS or RFS, or both. Performance improvements in both CPUs and networks should help this process be more cost-effective.

##### 4.2. Shared Environments [Collinson]

Sharing work means more than sharing data conveniently; it includes the use of maintenance tools (like SCCS), communications tools (like E-mail), and any other tools meant to help programming as a group activity. This attribute measures the efficiency of this activity.

		UNIX	Rest
	1974	5	0-5
Leverage	1978	5-10	0-10
summary	1982	5-15	0-15
	1986	5-20	0-20
	1990	10-30	0-30

- In 1974, UNIX had the basics for a good group programming environment <5>.

- By 1978, many groups had built coordination tools to support larger groups of people, SCCS was widely used, and *uucp* at least existed to provide some minimal communication facilities <5-10>.
- By 1982, a number of groups had built much more extensive programming environment systems (such as SOLID or MESA inside Bell Labs), and mailers had become more sophisticated <5-15>.
- In 1986, more powerful distributed system facilities are becoming available <20>, including some good non-UNIX ones by Apollo and DEC.
- By 1990, we can assume that the best UNIX-based group programming environments will be as good <30>, and maybe better, than the best available otherwise, mainly by catching up with the graphics that it does not yet support consistently.

#### 4.3. Bigger Problems

"We've Got Big Rocks and We Need More Hands"

Another way to measure leverage is to look at the maximum number of people that could work together using UNIX as a development base. This mostly derives from the size of machines and the internetworking thereof. The numbers below give approximate maximum sizes.

		UNIX	Rest
	1974	25	200
Leverage	1978	100	400
summary	1982	400	500
	1986	500	500
	1990	1000	1000

- In 1974, project size was pretty much limited to a group of people who could fit on one PDP 11/45.
- In 1978, a project could survive the use of several close-linked 11/70s. Some large projects simply could not fit onto the available UNIX boxes.
- By 1982, larger projects used linked groups of VAXen. Really large projects (like #5 ESS) were moving onto large mainframes, using UNIX/370.
- In 1986, UNIX runs on the largest computers. Even the largest projects can use UNIX for their development support.
- In 1990, I assume that truly immense projects can be UNIX-based if they so desire. I would hope that other areas of leverage have improved enough that we could do immense projects with fewer people, but I suspect that people will simply try to do even larger projects.

#### 5. Reusability of Code [Ritchie, Allman]

"New Rocks, Same Old Levers"

Not writing new code is the ultimate in programming leverage. Reusability has 2 separate aspects. First, there must be something to reuse. Second, you have to be able to find it.

- This is so subjective that I've omitted the numerical ratings. However, I'd observe that the common wisdom of 1974 was to re-read the UNIX Programmer's Manual once a month. You always learned something new, and besides, it didn't take very long to read it.
- By 1978, this was harder, and it's unthinkable today. Our ability to organize software and make it available has lagged behind our ability to write it, and we've yet to achieve Doug McIlroy's 1968 vision of software component catalogs. Another way to say this is to paraphrase Maslow's comment on tool-poor environments:

"To the person with only a hammer, all the world looks like a nail."

Unfortunately, in a tool-rich environment that has not kept its tools superbly organized:

“For the person with a giant workshop, it may take longer to find the right screwdriver, than to take the hammer, rip the screw out, and be done with it.”

- I hope to see substantial progress by 1990, using high-bandwidth interfaces and tools akin to Smalltalk browsers.

## **6. Portability [Johnson, Ritchie]**

This may be the most important area of leverage, but it is the easiest one to describe. Quite simply, no other operating system comes close to matching the portability of UNIX to different hardware architectures. Although UNIX has often sacrificed short-term performance for portability, it has often prospered in the longer run by being able to quickly move to new architectures. Consider how few other previously-existing operating systems have been able to shift to microprocessors. In fact, in the next few years, we're likely to see an ironic payback to UNIX in return for its portability. Most older operating systems are tied to specific hardware architectures, whereas the existence of UNIX makes possible the creation of new architectures without requiring huge efforts to develop new operating systems. Thus, most new machines will run UNIX, and many may run nothing but UNIX. If it happens that big jumps in performance and cost/performance occur via new architectures, then UNIX will have a substantial advantage in gaining access to especially effective machines. Many people believe that RISC-based microprocessors are going to offer such big jumps over the next few years. (Since I work for a RISC microprocessor company, I might be accused of bias on this!) However, many companies (I counted at least 10, offhand) are betting the success of major projects, or even their whole existence, on this possibility. Every one of these machines runs UNIX. This leads to one last prediction.

By 1990, the most cost-effective machines in the 5-50Mips range will all run UNIX or some closely-related variant.

## **7. Conclusion**

UNIX offers us various kinds of leverage. Internally, its leverage has improved over the years. Relative to other systems, it has often given us more leverage than most.

In 1987, I think UNIX most needs to widely incorporate the improvements in human interface that have appeared on personal computers. Next, we desperately need to regain the degree of reusability leverage we once had, i.e., where you easily could find and re-use anything there was.

Finally, UNIX portability will offer an exceptional source for leverage over the next few years, as people will be able to move masses of software to much more powerful machines. The result should once again show the wisdom of investing in that direction. UNIX is already a lever for moving large rocks; if we can't use it to move the whole world in 1990, we'll at least be able to tackle mountains and small moons.

## **Biography**

Dr. Mashey joined Bell Laboratories in 1973, joining the Programmer's Workbench department the same week it received its first PDP 11/45. He worked on various UNIX-related projects, including PWB/UNIX, the merger of UNIX versions that resulted in UNIX/TS 1.0, and UNIX-based applications. In 1983, he joined Convergent Technologies, ending as Director of Software Engineering. In 1985, he joined MIPS Computer Systems, where he helped design the MIPS R2000 RISC microprocessor, and is currently Director of Operating System Development.

## UNIX: The Data Forms

Eric Allman

Britton-Lee Inc.

We can start off by asking what "the data forms of UNIX" really means. Well, there are a few things it definitely does *not* mean. It does *not* mean inodes and u-dot areas. It does *not* mean details of the passwd file. Nor does it mean the information contained in the super block of a file system or the layout of a symbol table in an a.out file or the naming convention for high density raw tape drives in the /dev directory. All of these are relevant, all contribute in some way to our topic, but none of them are what we want to talk about.

Take a moment to ask yourself what *you* think is fundamental about the way data is represented and manipulated in UNIX. It's a hard question, but one we will try to answer here.

### Historical Perspective

It's hard to think of UNIX as a system well into middle age. It bears its years well, but the simple fact of the matter is that the first papers about UNIX were published in 1974, and most of today's speakers were using the system already or shortly thereafter.

At the time UNIX was developed, computers were in a totally different segment of society than we see today. Computers were expensive, in the quarter million dollar range, or else had minimal power and very primitive software — the PDP-8 comes to mind. As a result, computers tended to be available only in scientific groups or in business environments, which could justify the expense on the basis of improved productivity, depreciation allowances, capital investment tax credits, and so forth.

The scientific world was divided between the heavy metal types, typically funded by the big boys in Washington who prefer to remain nameless, running applications such as counting the number of electrons that can dance on the head of a supercharged ionized pin, and the lab types who programmed their PDP-8s to monitor the equipment in real time, said data to ultimately be passed to the heavy metal types.

The business types were using their computers for what we now like to sneeringly call "traditional applications": payroll, inventory, accounting, and so forth. For them, the transition from tabulating columns of figures by candlelight with a quill pen to tabulating those same columns under fluorescent light using a computer was easy — a change of style, perhaps, but not of function. Word processing was a concept that had been mentioned, but almost never actually attempted. Remember, this was an era when lower case letters were considered sufficiently redundant as to be expendable.

Both the business and the scientific worlds had pretty clichéd data. COBOL may have used pictures while FORTRAN used FORMAT statements, but it all added up to having that data start in column 17. This was okay, because these systems were not typically interactive, and drum cards made it pretty easy to get to column 17.

This was also the era of the access method. Who could forget the famous HSAM (Hierarchical Sequential Access Method), designed to be useful on magnetic tape, and ISAM (Indexed Sequential Access Method), not to mention HISAM, SHSAM, SHISAM, QSAM, BSAM, HDAM, HIDAM, and of course VSAM.

One of the reasons these tended to be so popular was that the datasets being used tended to be quite large as measured relative to the power of the processors on which they were run.

Today, searching sequentially for “foo” in the dictionary takes seven seconds on a VAX 750, a pretty feeble machine by today’s standards, but at one time even this 200K “database” would have been considered large. This same command takes just over one second on a VAX 8600 — a machine that is not destined to be considered “fast” for much longer. When one has a large dataset it makes sense to do a lot of work at the beginning of a query to set up the most efficient access path. But when the dataset is small, you might as well use brute force.

In those early days, our concept of “large” was a lot smaller than it is today, since the hardware was much less powerful. Also, only the “large” datasets were stored on computers, since the cost was prohibitive for “small” datasets.

To summarize: computers were either large, expensive, and geared toward very structured applications with elaborate access methods, or small, real time machines with feeble software.

Enter UNIX. Ken Thompson went to school at Berkeley, and had contact with Project GENII, one of the first timesharing systems built, based on an SDS 930 (changed to XDS during the tumultuous 60’s, and with hardware changes later known as the 940). This system was unusual beyond the timesharing aspects — one of the main language processors on the system was SNOBOL, and free format input seemed to be the style. After leaving Berkeley, Ken went to Bell Labs and worked on Multics, a joint project with GE.

Multics was a massive system, very much in the mode of the time — big expensive systems running on big expensive hardware (e.g., TSO/360). But of course, it had some nice ideas, and when Bell pulled out of the project it was difficult to go back punching cards and submitting your job to be completed, maybe, with same day service. The alternative — a bare box with minimal software — held little appeal.

Every great breakthrough has something about it that contradicts the common wisdom. I would argue that with UNIX, it was the realization that you *could* have a nice environment without a massive CPU, by observing that 20% of the services gave you 80% of the benefit. You need a scheduler, but you do not need a scheduler that allows the system operator to adjust your quantum, working set, and priority — for that matter, you don’t need an operator. You need the ability to access tape drives, but you do not need a tape allocation-authorization-accounting subsystem. You need a file system, but you do not need to build special-purpose access methods into the system. You need a way for programs to pass data between them, but it doesn’t have to be as complex as special access methods, data dictionaries, and so forth — something as simple as text will do.

### General Observations

Perhaps the most dramatic distinction of UNIX is the application of the “Small Is Beautiful” principle. UNIX started with small computers, and the presumption of small datasets. This led directly to the principle of small programs that did small tasks. Simplicity was everything. The UNIX *sort* program cannot sort multiline records or binary files, but the cost of supporting them was higher than the benefit achieved — and besides, the heart of the sort algorithm was available to the C programmer, who could sort any data type required. True, the file system didn’t permit a program to preallocate blocks or assign them to specific cylinders, but on the other hand, the documentation was only two pages, and it did everything these *simple* applications required.

One of the major areas of simplification was how most UNIX programs represented data. Nearly everything is represented as lines of text. For example, the `/etc/passwd` file is a simple text file; more conventional systems would represent this file as a binary database, and special tools would exist to update this database. There are advantages to the database approach — prevention of records with syntax errors, duplicate login names, or whatever — but again, the UNIX approach is simpler, and most of the functionality is maintained.

Let’s examine text for a moment or two. Text is normally associated with language, which reminds me of a lot of difficult grammatical rules I was never able to comprehend, which

seemed to revel in the exceptional cases rather than the simple realities. In the computer world things aren't that different — most compilers have a *scanner* as the front end, which has as its sole responsibility breaking the input text into tokens (that is, *words*) — and in many compilers this consumes a large amount of the total processing time.

On the other hand, there are a lot of *good* things about text. Most of all, it is familiar. Our terminals deal with text, we enter text, and most of the time we are presented with text. True, some programs take input from a mouse or palmprint reader, and some output to phototypesetters or voice synthesizers, but text is still comfortable and familiar, and many programs use this as their primary input/output paradigm.

One of the best parts of the familiarity of UNIX is engendered by the consistency that permeates the system. This consistency can seem clumsy on the surface, but becomes beautifully elegant when seen as a whole. This is very much like sublanguages such as SQL (Structured Query Language) used in many relational database systems. For example, the UNIX *who* program might correspond to the SQL program:

```
select * from wtmp;
```

Such languages begin to shine as more complex structures are created — tables can be correlated against one another without prior planning, using a consistent language. UNIX provides similar consistency using a much simpler language — but in turn loses the semantic consistency of a relational data sublanguage.

A system could standardize on an internal form that would be machine readable, and have a special “externalize” operator applied automatically when that data was printed. Smalltalk® uses this approach, and for many applications it works very well. Smalltalk data is always “tagged” in some way so that its type and identity is not in question, thus providing the information needed to print the data as well as to correlate it with data from other files.

UNIX prefers to use the external format as the exchange format. *who* outputs text, *grep* inputs and outputs text, *sed* inputs and outputs text, and thus a pipeline is built up. Pipelines would not be possible without a consistent representation, and UNIX just happened to settle on text. Even pathnames are text in UNIX — each component is separated by a slash, although an acceptable representation might be an array of strings. Internally, UNIX has to break up the pathname into a series of file names, and this takes time. But the alternative is worse. Imagine the horror that would abound if every program potentially had its own representation for naming subdirectories. Some of them probably wouldn't even let you reference files in another directory.

One of the chief beauties of using text as an interchange standard is that it is about the most portable representation ever devised. Integers have byte swapping problems, floating point numbers cannot even agree on the number of bits in an exponent, but text strings will live forever. Now, this may not be critical for applications that reside only on one machine, but the ability to link programs together across a heterogeneous environment is appreciated more each day.

Something about text makes it totally ordinary — it is special precisely because it is not special. Interestingly enough, the same thing is true of “special files” in UNIX. The thing that is special about a tape drive or a terminal or a line printer on UNIX is that it *isn't* special. Those of us who are getting on in years (that is, over 30) remember the days when a program that wrote to the terminal had to use totally different system calls than those used to write to a disk file; writing to a printer was an adventurous exercise. Some systems provided subroutine libraries to hide the differences, but *real* programmers, the type that would happily add three *gotos* to avoid one addition and laughed in the face of ALGOL, never designed to use them. Subroutines were too large and too inefficient for a *real* programmer to use.

UNIX preempted this attitude by preempting the plethora of system calls that lead to it. By choosing a sufficiently simple model, all devices can be reduced to fit that model. The

model can't work, because it is too simple, yet it does work, because it is simple enough.

For example, we all know that human beings are impossibly perverse. We make silly mistakes such as typing errors, and then get offended if we are penalized for it. UNIX handles these inconsistencies by localizing them into a single module interposed between the fallible human and the nice responsible program. Contrast this with other "small" systems that tend to leave this to the application program. The inevitable result is dozens of slightly different behaviors.

The simplicity that makes this work is multifold. First, lines are delivered one line at a time by simplifying the interface so that no device need ever return the full amount of data requested. Second, reads of less than a full line are buffered so that no data is lost. Third, characters that must be handled specially, such as carriage return or control-D, are handled by the low level driver, thus simplifying applications that deal with terminals.

Finally, options that simply *must* be handled, such as preference options, are implemented using special calls. Other systems might use exciting calls such as read-a-line-from-the-terminal-using-\$-as-the-erase-character — UNIX sticks to "read bytes", and you can change the erase character to "\$" if you wish.

Probably the ultimate simplification of all was that UNIX didn't really know about "text" at all, but just "bytes". All files were streams of bytes, without additional inherent structure. Thus, files containing text differed from files containing binary codes only by the bytes they contained. Thus, the *cat* program works equally well on text and binary files.

In fact, UNIX was actually extremely functional for the time. For example, when writing to a terminal, tabs were expanded automatically, necessary delays were inserted by the system, full duplex was standard, and of course, lower case characters were supported everywhere — in fact, upper case characters were a bit inconvenient.

UNIX also offered a full hierarchical file system. To those of us who were familiar with systems having file names shared across all users on the system, this was a godsend. Creating one's own directories without ever wrestling a VTOC (Volume Table of Contents) to the floor was amazing. Even the "big" systems didn't permit this, at least not without experiencing the vagaries of JCL (Job Control Language). The file system itself became a powerful tool for managing data, often embodied in other tools on other systems.

UNIX also provided reasonable protection. By "reasonable" I mean enough, but not too much. File modes could be set, but they always defaulted to "permit". Big computers had always defaulted to "deny" and little computers usually didn't even have a distinction between two different users.

Perhaps one of the most amazing things about UNIX was the large number of sophisticated text processing tools. Besides *nroff* and *troff*, *diff* has been around since the earliest versions (called *proof* in Version 4). It's hard to imagine UNIX without *grep*. Even *spell* existed as early as Version 5. All these tools performed the sort of "hard" jobs that are commonplace today, but which were considered too expensive for mere mortals at the time.

This philosophy is actually taken to an extreme. For example, programs are nothing more than text with some special semantics. All the text processing tools can be used. For example, to find all uses of a given variable, the *grep* command is used; contrast this to the bulky cross reference listings so ubiquitous on batch systems. (You remember the cross reference — it came out between the program listing and the symbol map on the chain line printer.) In fact, a compiler can be thought of as a highly specialized text processor, reading an input with certain rules, just as *nroff* reads input obeying certain rules.

### The Data Forms

Let's look at a few specific examples of how these principles are applied to the data forms of UNIX.

All files are modeled as binary streams of bytes on UNIX, be they disk files, special devices, or pipes. Compare this to a system like VMS, having several different file types. To correctly read and process a file on VMS requires careful consideration of the file's type and

attributes. The VMS guru in my office counters by saying the VMS is actually a superset of UNIX — he is correct — but, UNIX chose to implement the 20% of the code needed to handle 80% of the cases.

Because of this simplicity, UNIX permits you to apply almost any operation to any file. You can *cat* a directory to your terminal, although it does not make sense. You can submit *nroff* output as input to the shell, or *nroff* shell scripts, use *ar* on a *tar* file. These will all “fail” in different ways, some by giving an error message, others by putting your terminal into some inexplicable mode. Is this good?

In the sense that full generality is achieved, this is good. The *dump* program, which normally dumps to tape, can also dump to a disk file, a raw tape, or even to paper tape. Likewise, the “tape archiver” program *tar* is quite happy to archive to a pipe instead of a tape. An ordinary user program designed to read input from a terminal can immediately read input from a file — or from another program. You can program your function keys or access special terminal functions without using special files or system calls.

On the bad side, all over the world today innocent users are blasting binary files to their terminals, locking up their keyboards or generally leaving both themselves and their terminals in a confused state.

Despite the total generality of a binary stream, files containing text on UNIX are more tightly constrained. Despite the lack of record boundaries in UNIX files, a text file signifies records by using the in-bound “new-line” character. This one small decision has immense implications, that can be seen most easily by examining the alternatives.

A popular way of representing lines is as a variable length record. VMS represents most text this way. This method can be represented by the Pascal I/O system — *writeln* writes a complete line. There are no *major* problems with this, but on the whole I have to class it as “second best”. Unfortunately, programs that might prefer to treat the file as a sequence of bytes (such as *tr* or *wc*) do not have that option with record-based files, (bulk copies are still forced to notice line boundaries) and an annoying syntactic “burp” is added to the language.

Another common, and extremely obnoxious, way of representing lines is to end them with the two characters “Carriage Return” and “Line Feed”. I invite you to pause for a moment to consider the problem of finding the end of a line under this convention. Some programs have been known to do things such as read to a CR (Carriage Return) and then blindly discard the next character or read to a LF (Line Feed) and blindly discard the previous character. Some just discard CR and view LF as the “real” end of line character; others do the opposite. Just what does a lone CR or LF mean anyway? Does LF—CR mean the same as CR—LF?

The same philosophy applies to strings in memory. They are terminated by a null byte rather than a newline byte, but the principle is similar. The obvious alternative is to represent a string as a count followed by the characters. The obvious advantage of this approach is that all bytes can be included in the string, including the null byte. Against this, weigh the complexity and inconvenience of representing a string as a complex data type — instead of

```
char *s;
```

a more complex declaration is required:

```
struct string {
    int s_len;
    char s_val[];
} s;
```

Representing the base of a substring is no longer simply a pointer to that substring, but a pointer to the base of the string plus an offset. And forget forever the convenient paradigm:

```
while (*cp++ != '\0') { ... }
```

Null termination is compact and clear, but it does remove one character from the available character set.

Again, UNIX has made a sacrifice at the very lowest level, and in the process sacrificed some of the jobs it might otherwise have done. In exchange, it has gained a simplicity and an ease of expression. Of course, UNIX *can* implement counted strings, just as MVS *can* implement null terminated strings. But the overwhelming bias is toward one style or another, and UNIX has chosen the null terminated *modus operandi*.

Another example of the “simplicity is power” principle is the handling of argument vectors and program invocation. All programs are handled identically, with system programs handled identically to user programs. As obvious as this may seem today, it was not obvious when JCL was produced, allowing you to pass additional parameters only by using the cumbersome “PARMS=(...)” syntax.

### Prognostications

I have gone on at some length about UNIX, saying very little beyond the obvious, in an attempt to express that the obvious may be more important than it first seems. As a contrast, I offer some of the changes I might make were I designing UNIX today.

First and foremost, I would make the system more object-based. This would be reflected most dramatically in the file system. Each inode would have an associated type that would define its behavior. Inodes could be active, that is, they could be programs that would be invoked when the file was opened. This would replace “special files”, but would not result in “file types”, since the fundamental interface would remain a byte stream. Eighth Edition incorporates much of this concept, but is bound to maintain the fundamental interface, which I would prefer to abandon in favor of something more general. At minimum, this would require expanding the file “format” field from four to at least 16 bits. These bits would be an index into an “implementation” table, that could indicate kernel routines or user level processes, and which could presumably make use of the additional nodes available in many modern CPUs. In this sense, I agree with the “nugget” concept as proposed by MACH.

However, I would not go so far as to make absolutely everything be an object, despite the temptation. How nice it would be for *sort* to automatically identify the numeric fields and sort them accordingly! But the additional complexity, completely justified in a database system, is probably inappropriate for a general operating system, especially one as focused on text as UNIX is.

I think I would make some substantial changes to UNIX tape handling. This is the primary area where the beautiful transparency of UNIX develops a nasty wart. The main problem here seems to be that tapes are *inherently* record based, which UNIX simply does not understand. Sadly, I would probably put ANSI tape conformance into the system (albeit perhaps not the kernel) — despite its verbosity, that is what the industry is using, and the egocentric attitude of *tar* and *cpio* cannot persist.

### Conclusions

Systems succeed by virtue of providing an interesting set of services to their users. UNIX took an unusual approach, providing a small set of very powerful primitives, using the simplest possible model of data at every turn. This resulted in an interesting combination of the very powerful and the almost absurdly simplistic. Consider the power of the hierarchical file system in contrast to the simple model of files as byte streams; the power of device independence against the simplicity of the null terminated string. The power of the of the data forms of UNIX is none of these things individually — but it is the gestalt that results from their juxtaposition.

## Annotated Bibliography

C. J. Date, *An Introduction to Database Systems, Volume 1*, Fourth Edition. Addison Wesley (1986).

This book gives an excellent introduction to database systems. In particular, the discussions of access methods in various systems are germane to this topic.

E. I. Organick, *The Multics System: An Examination of Its Structure*. MIT Press (1972).

The Multics system is the immediate ancestor of UNIX. Although Multics is much larger and more complex than UNIX, many of the UNIX concepts owe their genesis to Multics. The comparison, particularly features deleted in UNIX, is interesting.

Richard F. Rashid, "Threads of a New System." In *Unix Review* 4, 8 (August 1986).

This article describes the MACH system being developed at Carnegie-Mellon University. Although the user interface is intended to be fully compatible with Berkeley UNIX, the kernel is being completely reimplemented. MACH intendsto use the *nugget* concept, in which many services provided by the UNIX kernel are moved into user-level daemons.

D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System." In *Commun. ACM* 7, 7 (July 1974).

The seminal paper on UNIX. This paper describes the system in a manner that conveys the beauty resulting from pushing simplicity to the extreme.

*UNIX Time-Sharing System Programmer's Manual* (Research Version), Eighth Edition, Volume 1. AT&T Bell Laboratories (February 1985).

Eighth edition continues the tradition of unifying principles wherever possible. For example, *streams* (a generalization of the "line discipline" concept) were first developed for this system (later incorporated into System V). The concept of "file system type" is added and used to implement such diverse ideas as network file systems and the ability to debug a running process.

*UNIX Programmer's Reference Manual*, 4.3 Berkeley Software Distribution (April 1986).

Most of the concepts discussed in this paper are described in detail in this document.

## Biography

Eric Allman first encountered UNIX at the keyboard of an ASR 33 Teletype connected to a PDP 11/45 running UNIX Version 4. Since then he has used Version 5, Version 6, Version 7, PWB, System III, and System V, and made major contributions to the Berkeley Software Distributions. He has worked in the areas of database systems, electronic mail, text processing, and general system management.

He is currently a Staff Programmer at Britton Lee, working on user interfaces to relational database machines.

## UNIX: The Language Forms

Stephen C. Johnson

Dana Computer, Inc.

From the earliest days, UNIX has been a very good host for languages, and encouraged their growth. The resulting proliferation of languages included "conventional conventional" languages such as C and FORTRAN, "unconventional conventional" languages such as *awk* and *sh*, and just plain unconventional languages such as *eqn* and *troff*. This language growth was fertilized by device independent I/O, which carried the linear character at a time view of the world to its logical extreme. Tools grew up that allowed conventional and unconventional languages to be built and changed quickly. Our future challenge is to find an equally rich set of models that can deal with two dimensional input and output, mice, icons, and the other trappings of modern computers.

### The Early Days

During the early 1970's, UNIX and C evolved in an unusually synergistic fashion. Although the earliest versions of UNIX were written in assembler, the desire to rewrite UNIX in C had considerable influence on the early design of both UNIX and C. C evolved from a typeless language, B, which in turn had been strongly influenced by BCPL. B had a short but vigorous life at AT&T in the early 1970's; there were B implementations on both the GE/Honeywell mainframe and on the early UNIX systems.

UNIX B was implemented as an interpreter; unfortunately, there were a number of features that made it difficult to compile B efficiently on the PDP-11. All B objects were treated as being the same size — 16 bits. This made it difficult to deal with bytes (which had to be manipulated with function calls). A more subtle problem was that pointers (alive and well in B) could be made to point to the next 16-bit object by adding 1 to the pointer. This meant that pointers had to be word addresses, not byte addresses, and every pointer had to be shifted before it could be used by the machine.

While B was a nice vehicle for user software such as editors and *yacc*, it was hard to consider writing an operating system in an interpretive language. Another driving issue was the desire to deal with objects of different sizes; types arose to keep track of the sizes of objects, rather than for any desire to have types *per se* to keep up with other languages (the virtues of type checking were discovered rather late in the development of C). In turn, the big push for objects of different sizes was the desire to describe structures.

Once the notion of different sized objects was added to B (converting it to C), structures could be added, and the language could be compiled. It is interesting to note a number of ideas from B that have survived in C. Not only is the syntax very similar, but one still adds 1 to a pointer to get the next object, just like B. Also, C's rather questionable treatment of arrays (turning them into pointers when passing them to functions, for example) comes straight from the way that arrays were treated in B and BCPL.

Later, when V7 UNIX was written, with portability as a strong consideration, C evolved further to allow portable ideas to be expressed more portably (the constructs *sizeof* and *union* are the most visible result of these pressures). The desire to keep operating system code efficient, and the difficulty of incorporating runtime code for I/O and heap storage in the kernel, kept C small and, above all, efficient.

In more subtle ways, C influenced UNIX. Although the earliest manuals contained assembler interfaces, these were soon dropped, and the C library calls became the official system interface. By V7, major system structures were described and accessed through C header files; these files represent a major challenge to those who wish to write system code in languages other than C (e.g., Pascal, Modula, Ada, ...). Data representations such as C's null-terminated strings have become *the* way to talk to UNIX, to the discomfort of other languages. Finally, C style, from lexical conventions to operators and syntax, has affected many UNIX commands and languages, most notably the Berkeley C Shell.

This close philosophical tie between the language and the system had some other consequences. It was natural for the designers and early users of UNIX to communicate with the system through specialized languages. For example, linguistic notions such as regular expressions showed up in many early UNIX commands (unfortunately in a somewhat irregular manner). These concepts are simple for programmers to master, but are likely to be a bit out of the experience of an accountant or secretary. But, in those days, UNIX was only for programmers.

Early access to UNIX was almost entirely through character at a time terminals, mostly mechanical, limited to 10 or 15 characters a second, and printing on paper (even "glass teletypes" didn't have upper and lower case capability, which was supported in UNIX from the earliest days). Those who never lived with these machines cannot imagine how it was possible to get anything done. For example, the bits on a single Macintosh screen would take almost a half hour to send at these rates. Much of the fabled "terseness" of UNIX was simply self-defense against the leaden unresponsiveness of slow terminals over slow lines, and regular expressions allowed one to do a lot of work without having to hit too many keys.

### Radical Notions in Compiling

It is difficult to convey the shock and dismay that many early C users felt when they found that the C compiler did not generate any listings! The preprocessor was a separate program, the two compiler passes were separate programs (and later, the optimizer as well), and the assembler and loader were separate. If one thinks of these as "job steps", following the philosophy of the times, the shock of this approach may be more understandable —such a way of compiling on TSO would have required at least 50 or 60 JCL cards, and probably more if all the special cases (e.g., replacing an existing object file) were properly handled. In fact, in many systems the preprocessor could not have been written, because there was no way to open named files from within a running program.

This division of the compiling job into pieces has continued to this day. When C portability began to be taken seriously, a program, *lint*, was written that would do strict type checking and test for consistency of programs between files. C together with *lint* actually provided a very strongly typed language, rather unconventionally constructed. Program generators such as *Yacc* and *Lex* wrote C programs, and let the compiler do the hard parts. This philosophy continues in the implementation of successor languages such as C++ and concurrent C, which are implemented as preprocessors that generate C.

In a similar vein, it was very unusual for the time to have compilers that were not part of the operating system. A compiler could be written or changed without requiring the rebuilding, or at the very least the rebooting, of the host operating system! Libraries could be created and edited by users, not systems administrators! Will wonders never cease! Actually, these ideas were not new in Bell Labs Research, where similar things had been done on systems dating back to the IBM 7094, but for most of the rest of the world they were pretty radical.

C function calls were fast! Unlike FORTRAN or PL/I, one could use lots of small functions, and design in a clean, "modern" manner. Well, it turns out that C function calls are not all that fast after all, especially on the PDP-11, but we all believed what we wanted to believe and went forth and did good works. Some dynamic measurements have shown that many C programs do a function call or return every couple of dozen instructions; this is far

more frequently than in normal FORTRAN or PL/I programs. This pervasive style implies that C programs run rather poorly on machines where, for whatever reason, the calling sequence is slow. In environments where there is a "system standard" calling sequence, allowing C to talk to other languages (a good thing!), this calling sequence is often oriented towards other languages with different characteristics, and C efficiency suffers. Similarly, the hardware CALL instructions are often far more ornate, and far slower, than C requires.

### Portability

By now, only a small fraction of all UNIX users ever used C on a PDP-11. Effective portability is perhaps the biggest single factor in the success of the UNIX system. However, the portability of C and the portability of UNIX are very closely intertwined. UNIX portability would be difficult or impractical were it not written in a portable, efficient implementation language, a language that itself was portable almost from its inception. Conversely, C has spread from 8086's to Crays in large part because it is immediately available on UNIX systems over a wide range of machines.

The application designer writing in C can be sure of finding many target systems of all shapes and sizes; this further encourages the spread of C. In fact, application portability encouraged the first port of UNIX; I remember Dennis Ritchie remarking that it would probably be easier to port UNIX to another machine than to port a large application to another operating system.

### Lots of Other Languages

An experienced UNIX user probably knows a couple of dozen languages, some simple, some quite complex. Besides the conventional languages, such as C and FORTRAN, there are unconventional languages on UNIX, such as *awk* and *bc*, that apply standard language principles and techniques in slightly unusual settings. The shell has become a first class programming language in its own right, and probably one of the highest level languages in common use (its basic data objects are files and processes, which is about as high level as you can go, and parallelism is an effective, integral part of the language design).

However, UNIX has many unconventional languages and language forms as well. In these cases, ideas and techniques from standard programming languages are applied in tools far removed from programming. Frequently, the UNIX tool for a particular job is much more "linguistic" than similar tools in earlier systems. For example, regular expressions, in various guises, are incorporated in the shell, in editors such as *ed*, *sed*, and *vi*, and in search programs such as *grep*, *fgrep*, and *egrep*.

In the text processing arena, *troff* and *nroff* are true programming languages, albeit not models for style or design. On top of *troff*, other languages such as *eqn*, *tbl*, *pic*, and *refer* have flourished.

Another very interesting language is represented by *make*. Unlike most of the above languages, *make* does not use a description of a process or actions to be carried out; rather, it is similar to a constraint-based system; the makefile tells *make* what the desired final state is to be (e.g., *x.o* should result from compiling *x.c*) and the *make* program analyzes these constraints and decides what actions are to be taken.

### Language Tools

Two other languages, *yacc* and *lex*, were available to encourage the spread of these languages on the UNIX system. Using these tools, a conventional language interface, even one of considerable complexity, could be prototyped in under a day, and, what is more important, could be cleanly and safely extended and changed in response to the needs of early users. The underlying algorithms were efficient enough, and the programs well enough engineered, that their use spread even to the shell.

That these tools could be written at all owes much to the state of computer science theory at the time, which provided the models and techniques on which the tools were based. Another positive influence was the organizational structure of Bell Labs, which intermixed a strong theory group with the UNIX pioneers, and encouraged their mutual interaction.

### Abstraction Levels

Since many of the tools available on UNIX took text files as input, and produced text files as output, it was not long before these tools were combined to raise the abstraction level from the particular to the general. Some of the earliest examples of this are shell command files: a frequently repeated set of commands were captured in a file, marked executable, and a new command was born. Similarly, *ed* and *sed* scripts were frequently used to perform similar editing functions on many different files. *troff* macro packages serve a similar function, providing common features across many different documents.

A mode of behavior frequently seen in the early days was doing an *ls* command into a file, to capture a list of file names, editing the list of files to remove (and possibly add) some, making the file list into a command file by editing one or more commands, and finally feeding the resulting file to the shell. This mode of behavior made very efficient use of the slow bandwidth available between user and system, but it was error prone, the errors could be very damaging (e.g., files getting wiped out), and the way of thought was alien to most casual users of the system.

### Device Independence

As mentioned earlier, many of the UNIX tools were based on a very high level data object—the file. UNIX Files were not just for simple disc storage of data; they could also be used to read and write terminals and tapes, control phone calling units and voice synthesizers, access kernel memory, and talk to special hardware (such as a chess playing machine).

Device independent I/O, and the resulting strong support for a linear file structure, is one of the biggest strengths, and biggest weaknesses, of UNIX. The strengths are obvious: simplicity, generality, and wide applicability of tools. The weaknesses are the weaknesses of every simplification: the model fails to capture increasingly important aspects of reality, and is in consequent danger of falling into disuse. As cpu/user bit rates increased, and cycles became cheaper, new modes of user/machine interaction became first possible, then inexpensive, then desirable, and finally nearly universal.

The first major stress on device independence came in the late 70's, as glass teletypes with 24 by 80 character resolution became available, faster, and ultimately cheaper than mechanical terminals. New editors such as *vi* and *emacs* were developed to take advantage of higher rates of interaction and the 2D nature of the the screen. Cursor motions replaced global replacements as the dominant style of editing. As a consequence, *ed* and *sed* files fell into disuse; it is hard to imagine a similar notion for *vi* or *emacs* input. The saved file of cursor commands and keyboard key presses does not really capture the experience of using the editor. What abstraction mechanisms there were (e.g., programmable function keys) were incorporated into the editors themselves.

Bit rates continued to increase, costs continued to fall, and new modes of interaction became possible and attractive, most notably bitmapped terminals, mice, windows, and icons. These are potentially very accessible to the casual user (vis. the Macintosh), but, as with the editors, allow for much less opportunity for abstraction within the traditional forms (can you imagine a shell command file on the Macintosh?). Once again, some of the slack was taken up by editors (*emacs* macros, for example, are looking more and more like command files).

More seriously, the device independent I/O, based on a linear array of bytes, has almost broken down in dealing with these forms of user/machine interaction. The *curses* package is slow and limited. Writing programs that drive a modern terminal as a stream of bytes is difficult and unnatural. Mice are nice, but hard to trap in a stream of bytes. \$1000 personal computers provide graphics, color, and interaction seldom found even in very

sophisticated and expensive UNIX systems. And with this breakdown, traditional left-to-right languages are falling away. Mice and icons don't work that way. And the tools that made these languages, *yacc* and *lex*, are being nibbled by mice.

### The Languages not at the Party

There are thousands of programming languages; most are not on UNIX. FORTRAN is a poor stepchild on most systems, since FORTRAN programs tend to use floating point (indifferently supported in areas such as the signal mechanism) and to be an order of magnitude larger than C programs, leading to loading and debugging in geological time. COBOL has almost no area of intersection with the concerns of UNIX users; the things that COBOL does well (interactive forms generation and processing large amounts of data) are things that UNIX systems typically don't do, and, worse, stigmatize as unimportant. LISP has a vocal following, but many of the nicest LISP environments are hard to port to UNIX because, as mentioned above, there is inadequate coupling between the display and the program to provide a truly attractive environment. Put another way, some of the strengths of LISP are weakened, and many of the strengths of UNIX are irrelevant to LISP applications. Smalltalk® and other object oriented languages seem to require environments dedicated to their care and feeding in order to provide even the illusion of acceptable performance. Finally, database query languages and fourth generation languages would seem to have every reason to flourish under UNIX, but (perhaps because of a shortage of database programs) they have not become dominant on most systems.

Just as the success of C and UNIX are closely linked, some of the failure of these languages to prosper can be laid at UNIX's door. Many of the programs written in these languages are designed for people quite different than UNIX Guru's —bank tellers, physicists, executives. What does it profit me to write a LISP program that talks to a craftsman in natural language, when the operating system says (as mine did this morning) "failure to reboot due to inconsistent file system: run fsck". A message beginning "panic" is likely to be taken as a command to the user. These languages will not flourish under UNIX without some attention to these system issues.

### The Future

Perhaps, as one convert enthusiastically said, "UNIX is the FORTRAN of the eighties". It is equally likely that it will be around past the turn of the century. But, like FORTRAN, it may be viewed as having Alzheimer's disease long before it gives up the ghost.

What abstraction, if any, can replace files as the major common data structure for future systems? Files just don't work any more, and UNIX is now behind in the same area, interaction, where it once led. Will the shell, as we know it, survive as a programming language, or will it become object oriented or constraint based? Can UNIX adapt to higher bitrate user/machine interactions? What tools are needed to cope with applications based on mice, button, and icons? Will UNIX itself adapt, or will it be replaced? In summary, the future of languages on UNIX, and indeed on other conventional systems, reduces to an age-old problem: building a better mousetrap.

### Biography

Stephen C. Johnson received his A.B. in Mathematics from Haverford College and his M.S. and Ph.D. in Mathematics from Columbia University. He began working for AT&T in 1967, eventually heading the Computer Systems Research Department, Bell Laboratories, and the Language Development Department, Information Systems. During his 19-year career there, his research work included compiler construction, parsing, code generation, complexity theory, psychometrics, computer algebra, and VLSI design. His UNIX experience includes the writing of pcc, yacc, lint, early versions of spell, the port of UNIX to the Interdata 8/32 (with Dennis Ritchie), and more. Dr. Johnson is currently Director of Programming Languages for The Dana Group in Sunnyvale, California.

# UNIX and Networking: A Separate Peace

John R. Mullen  
Communication Machinery Corporation

## ABSTRACT

"Peace - noun. In international affairs, a period of cheating between two periods of fighting."

—Ambrose Bierce, *The Devil's Dictionary*

The rise to prominence of the UNIX system in the middle 1970's coincided with a particularly active period in the development of computer networks. Given the common interests shared between the UNIX and networking communities, it was natural that a close relationship develop between UNIX and networks. However, this relationship has always been, at best, an uneasy but usually workable truce, punctuated with bursts of both great joy and loud acrimony. The reasons and results of this tempestuous co-evolution will be evaluated for the purpose of understanding where we are, how we got here, where we might rather be, and how to get there, as told by a first-person participant.

---

John Mullen is currently a Senior System Engineer with Communication Machinery Corporation, focusing on high-performance network front ends for Unix systems. He received a B.S. from the University of Illinois at Urbana-Champaign in 1973. As a member of the University's Center for Advanced Computation, Mr. Mullen participated in researching distributed systems and distributed data management techniques using the Arpanet. At the Mitre Corporation, he concentrated on issues relating to use of TCP/IP on local area networks. He has been an active member of the Unix community since 1975.

## UNIX: The cult

*Peter Collinson (local name)*  
*AS/103/108/121/110 (Galactic designation)*

Galactic Cult Investigation Team,  
Canterbury, Kent  
(Small Island off Continent 3, Sol 3)

### ABSTRACT

Notes from some recent archeological findings on the birth of the UNIX cult on Sol 3 are presented. Recently discovered electronic records have shed considerable light on the beginnings of the cult. A sketchy history of the cult is attempted.

### Background

The UNIX cult is widespread across the Galaxy now and the surprise discovery of some ancient files in the archives of Intergalactic Brain Machines on Sol 3 triggered the dispatch of an inter-disciplinary investigation team. The files are extremely extensive, occupying all of a small island off the coast of Continent 3. It transpires that the island was taken over by Intergal in the aftermath of the Corporate Wars which plagued Sol 3 some centuries after the birth of the cult.

The team were asked to find out the original meaning of some of the incantations used in UNIX religious practice and also to shed some light on what it all meant at the start.

We should take this opportunity to use the ancient prayer:

UNIX is a trademark of AT&T in the USA and other countries.

Earlier versions of this prayer do seem to exist, it is unclear why the form of words altered. 'AT&T' was the Corporation where the Creators of the cult worshiped. The Corporation totally disappeared in the wars and many of its original records were either destroyed or altered by the victor in an attempt to 're-write' history. The placement of the country USA on the four continents has been lost.

### The Gurus

There seems to be still no trace of the original Creators of the UNIX cult, so we start our examination of the records with a group calling themselves the *Gurus*. The etymology of this word is not quite clear but it does have associations with religious teaching and the High Priests of UNIX are given that title today.

Extract from electronic phone tap of someone nicknamed 'dsw', believed to be Daniel Stuart Wilson:

"Of course, we were all isolated in the Version 6 days. When we changed things in the kernel we were on our own. There was no-one to phone to scream for help, in many cases there was no-one on the same site who you could discuss the problems with. We just had to get down and read that C. Sorting things out yourself was painful but you benefited in the long run. You became a better software engineer. After a bit you became a Guru and could lead others. This is largely

bluffing — given a new situation and a knowledge of UNIX you could extrapolate the problems easily and seem to be very clever. It didn't actually matter what you had or had not done, it mattered that you *appeared* to have done a lot and could talk confidently about RK05's, PDP-11/45's, typing *chdir* rather than *cd* and a myriad other things which showed that you were a Guru."

This extract shows clearly that the cult grew in small pockets across the globe. In each centre, a few individuals were given the task of installing the paraphernalia of the UNIX cult and of converting others to its use. In the beginning, it seems that these individuals had little or no contact with each other; curiously, this appears to have strengthened their ability rather than weakening it. Other extracts from the archive indicate that the early practices of the cult were small and simple making it easier for one person to grasp their full meaning. As the cult grew, the practices became more complicated and the understanding harder.

The term *software engineer* refers to a person whose task it was to feed instructions or **programs** into the primitive versions of the Overlords which were extant at the time. Judging from the emphasis made in the records about the need to generate 'correct' programs, this was obviously an artistic task requiring considerable expertise. The reference to 'C' implies that there was some official form of speech or perhaps a special religious language which was used to convey instructions.

It is not clear whether the 'RK05' and the 'PDP-11/45' were the names of the Overlords or some ancillary equipment associated with them. However, the word *cd*<sup>1</sup> meaning 'to move from place to place' is still used in some arboreal societies on the planet US/115/110/105/120.

### The User Group

In current UNIX religious practice, the term *User Group* is used to refer to the congregations at the Hologram Services. It seems that after a while the early practitioners of the cult began to have meetings (where people actually appeared together in the same room). Why these were called 'User Group' meetings is unclear, since the early meetings were attended by Gurus and not by Users. At the time, a *User* seems to have been a derogatory term for the un-initiates. However, the head Guru at a site was given the title 'Super-User'; perhaps this was to hide the evangelical nature of the Guru's task.

The reason for the early meetings was mostly to allow Gurus to inform each other how best to perform religious conversion and how to get the most converts the fastest by *improving the Service to Users*<sup>2</sup>. As the word spread, the meetings grew in size, expense and quality of surroundings. They proved to be an exceptionally good way of upgrading low level Novices into higher level Gurus. This is a reflection on the early religious books which were aimed at Gurus and were often way above the head of the Novices. Novices were encouraged to attend the meetings to gain by word of mouth what they could not gain from the literature. At these later meetings, the nuts and bolts of UNIX practice were still discussed because many of the Novices were either acting in a support role to Gurus or planning to become evangelists themselves. After a while, the Gurus got bored with discussing '*inodes*'<sup>3</sup> and other topics began to creep in.

The meetings gradually altered in character, with the Gurus attending because they wished to see the other Gurus; and more and more Novices attending in the vain hope that they would learn something. The cult had spread so far by now that it was profitable for Corporations to become involved in the selling of UNIX paraphernalia and religious goods. Initially, these *Products* were advertised widely at the meetings. However, the Corporations often sent attendees with prepared scripts who were sometimes not even Users and who had

<sup>1</sup> Pronounced *see-dee*

<sup>2</sup> This is a contemporary term.

<sup>3</sup> The meaning of this is totally lost.

little or no knowledge of intricacies of UNIX practice. The Gurus and Novices were dismayed.

On their side, Corporations began to see that the word which they were spreading was falling on somewhat stony ground. This gave rise to the Great Split with a rival organisation being set up primarily aimed at selling and the original User Group concerning itself more with the cerebral activities of the cult. In the end, this was a good thing because the Gurus were able to start deriving benefit from the meetings again since there were now spare slots for educational talks. In fact, the rival organisation was wiped out in the Corporate wars because it had allied itself strongly with AT&T.

The User Group, then, provided some important functions. It supplied a forum for discussion of the practices of the cult. It provided a meeting place for the widespread Gurus who initially met to discuss their work but as time passed they went just to meet each other. It spread the word to Novices; and as the cult grew, it provided a place where new ideas on the direction which things should take could be discussed.

Another more hygienic method of worldwide communication grew out of the cult with the formation of the Network. This is discussed in the next section.

### The Communicators

When the cult had grown to worldwide proportions, we begin to see the emergence of a global communication system — the 'Network' or 'Net'. The activities on the Network can be deduced from the electronic archives but the material is so vast that scanning it for relevant information is proving difficult.

The majority of traffic on the network seems to have been communication between Overlords describing various error conditions. Here is a sample:

*<Various repeated unintelligible lines>*

From: MAILER-DAEMON (Mail Delivery Subsystem)

To: <uucp@a4los.uucp>

Subject: Returned mail: Service unavailable

----- Transcript of session follows -----

>>> DATA

<<< 554 sendall: too many hops (30 max)

554 <megd>... Service unavailable: Bad file number

----- Unsent message follows -----

*<More of the same>*

There are very many other examples of the same type of message. However, these messages do place things in context — we now know that the communication system was called *Mail* and from this we infer that the mechanism was intended to permit communication between people. In amongst all the Overlord messages we do find some files which appear to emanate from one person and be addressed to another. A high proportion of the sampled files were intended to probe the capabilities of the Network, often provoking an Overlord error message. We know this because the destination address is the same as the source; in some cases the subject is 'Just testing' or something akin to that. This technique was perhaps used to investigate the ability of the Network to transfer files. We are left with a much smaller number of what might be termed 'useful' messages. In many cases, these consist of personal trivia showing that the Network was supplying the useful social function of allowing people to make and maintain contact.

Other messages consist of hieroglyphics containing many braces '{' and brackets '('. This type of message shows some form of regularity in structure but syntax analysis is hampered by the presence of so many exceptions. It is possible that these messages contain portions of religious ceremonial or it has been suggested that this is the ritual language 'C' and the exceptions are what were called 'bugs'. This last suggestion (by a post graduate student on the team) has been greeted with a certain amount of derision.

Mail messages have a recognisable format and can be distinguished from another type of message which occurs in much greater volume. It appears that these messages are part of a system called 'The News' broadcast to many sites. The News permitted individuals to send a single message to many people across the world. A sample of the contents imply that the News took over as the main method of spreading the UNIX word when the User Group meetings ceased to be totally useful in this role.

Judging from the beginning of many News files, the News was split into many separate subject headings. Over large periods of time, we see the subjects come and go with abrupt changes in title occurring from time to time. The subjects do not seem to be confined to discussion of topics with direct relevance to the cult. There are many headings which just carry 'talk' on various subjects. An analysis of the topics is being prepared as a background document since they fall outside the remit of the investigation.

It might be thought that the News would provide an excellent vehicle for the Super Gurus who must have existed by this time — but far from it — very few messages contain what might be termed confident information. Most seem to carry opinion which is contradicted in later messages. The evidence here is that most authors were Novices. The Gurus had either lost interest in spreading the word, or were simply too busy to wade through what must have been daily oceans of verbiage. It is also possible that Gurus just used the Mail to communicate, perhaps not wishing to impose their definitive opinions on the discussion with the thought that discussion is a healthy academic tool.

The Network spread slowly over most of Sol 3 with some areas being exempt because either they resisted conversion to the UNIX cult or were deliberately omitted for ideological reasons. A single News message from 'kremvax', an otherwise silent site, seems to have been met with a storm of protest. This shows that the Network crossed political boundaries and proves the contention that Sol 3 was split into separate economic entities predating the rise of the Corporations.

### The Vendors

The Corporations who were involved in the propagation of UNIX paraphernalia and religious goods were known as *Vendors*. The Vendors had a different view of the world from the Gurus, and this difference led to many schisms in the cult. The Vendors and Gurus tried to maintain a separate identity at all times. For example, when the Vendors attended the User Group meetings they provocatively wore different religious vestment from the Gurus. They did this to make it easy for other Vendors to identify them and to allow their easy differentiation from the Gurus and Novices.

At the centre of the clash of opinion was a fundamental difference in the perception of 'the User'. The Vendors were always complaining that UNIX religious practices were not 'User friendly'. By this, they meant that the act of worship was hard to learn, and the rituals were cryptic. They wished their Users to only deal in simple concepts and never to learn more than the basics of the rituals. The Gurus objected to this view because they believed that the learning of cryptic rituals allowed the worship to proceed faster and that limited exposure to only the simple concepts restricted the Users in a way which was not desirable. The Vendors believed that the Users were fundamentally stupid and without any hope of redemption; and the Gurus believed that the Users were fundamentally stupid but might be saved given the correct tuition.

The early Vendors were peopled by Gurus who had often to undergo the necessary clothing transformation<sup>4</sup> to demonstrate that they had switched camps. These Vendors were responsible for the spreading of the cult to a much wider and naive User population. Some of the Vendors tried to set up their own breakaway cults to avoid the central control imposed by AT&T, the Corporation where the Creators worshiped. These attempts failed. Other

<sup>4</sup> Often referred to as *mv tie neck*. This is inexplicable at present.

Vendors created Sects which specialised in worshipping the many minor Overlords which had appeared about this time. These Sects were partially successful and converted Users to their way of thinking. The Sects created by this method had special names related syntactically to the word UNIX. Only XENIX has survived as an example of this.

Then seeing all this activity, AT&T decided to become a Vendor.

The Gurus were horrified when the Marketing Staff, regarded today as the front line fighting force of all Corporations, were put in charge of the development and promulgation of the cult. The Creators were no longer allowed to directly influence the development, instead they were to pass their ideas to the Marketeers who would decide what was acceptable or not. The Marketeers spent a lot of time producing a new religious tome to help to guide the developers. The book was known as the *Svid* and laid out the central tenets of the practice. The Marketeers were determined that the *Svid* should be elevated to the level of all the other religious books. To this end they tried to make the *Svid* as cryptic as all the others, and succeeded.

After some initial shock, the Vendors accepted the central control which AT&T imposed because they saw that it made their Products accessible to more Users. 'Consider it Standard' became the watchword in a Crusade designed to eliminate the undesirable Sects which wished to differ from the *Svid*.

The Users of the Vendors Products were certainly pleased with AT&T's decision to become a Vendor. It meant that they were no longer tied to the Products of one particular Vendor but could pick and choose without altering their worship. The Gurus were less pleased until they realised that AT&T were unable to change the cult to eliminate them. Every Overlord where the UNIX cult was practiced still required a Guru to perform essential tasks.

### The Sects

The UNIX cult was always noted for its propensity to split into separate Sects. In the early days, the Creators had a release policy which made sure that all cult members possessed all the relevant facts in order to fully comprehend the implications of the worship. It was said that Users aspiring to be Gurus needed all the facts because the religious books were written for Gurus, and Users could make no sense of them. Unfortunately, access to the information was much abused because the Gurus immediately used the knowledge to alter things and many minor and major Sects of the Cult sprang into life. The ability of each site to generate its own Sect was somewhat curtailed by the cunning ploy of re-issuing the rituals and practices in a slightly altered form from time to time. The Gurus were soon tired of altering the same things every time a new set of rituals came through the door and they began to leave things alone. Also, by this time, the Vendors had made an appearance and because they believed in the *Svid* Crusade, they stuck with the orthodox mainstream AT&T view of the cult.

Even so, at the end of the period being researched there seems to have been two Sects with differing practices and rituals. The main rival to the orthodox view was a Guru lead Sect called the 'Berkeley System Devotees'. This had sprung into existence in the early days of the cult, taking advantage of the knowledge imparted to them by the Creators. However, the early Berkeley Gurus cleverly distributed their rituals and practices in the much the same way as the Creators and this ensured a wider following. The prominence was noted by a higher power and they were chosen to master the revisions of the Cult which were needed to permit worship on some new Overlords.

These new Overlords had managed to conquer the restrictive memory sizes which plagued many of their forbears. It was said by many that the new Overlords had not got this right but at least they did it. As a result the new Overlords had the potential to allow bigger and more expansive practices. The Berkeley Gurus grasped this opportunity with the objective of creating 'The Perfect Ritual'<sup>5</sup>. The Perfect Ritual was defined as one where all the letters of

<sup>5</sup> The rival Sects referred to this as 'creeping featurism'; the precise meaning of this obscure phrase is under investigation.

the alphabet were used as a 'parameter' specifying a distinct action or phase in the worship. The Berkeley Gurus never managed to create the Perfect Ritual, but they came close.

The Berkeley Gurus distributed their rituals and these became popular because the Marketeers inside AT&T were so busy creating the Svid that they failed to notice that Berkeley practices were slowly being adopted on all the new bigger Overlords. The Berkeley rituals were also liked by Gurus because they were nearer the 'Old Religion' laid down by the Creators. In a fit of pique, the AT&T Marketeers decided that they would no longer support the new Overlords and branded the Berkeley Gurus as heretics.

As heretics, the Berkeley Gurus decided to go one step further in altering their Sect. They proposed and executed a fundamental change of direction which was to become a 'De-facto Standard'. The new Sect was revolutionary because it allowed Overlords to talk to each other, but to do this a whole new litany had to be created. New words entered the vocabulary and new concepts were introduced. For many, worship in the new Sect was slow and unwieldy in comparison to what had gone before. But the new practices meant the easy ability to interconnect Overlords and this was demanded by the Overlords themselves.

Unfortunately for the participants in the Svid Crusade, many Users actually insisted on being able to use some of the Berkeley rituals. The pressure from the Users was such that we begin to see Vendors announcing their Products as being 'with Berkeley enhancements'. Finally, the AT&T Marketeers were forced to incorporate some features of the rituals which did not conflict with the teachings in the Svid. At this time, we also begin to see specially created Overlords which could be used to worship in the practices of either Sect simultaneously, this was known as the 'Universe Concept'.

The Berkeley Gurus were so broken by the gestation of the new Sect that many left, some to worship the sun and some to seek salvation in the noble task of Pixel Creation. It was thought that the new Sect would be the last to sally forth from hallowed halls of Berkeley because the staff were demoralised and without joy. The Svid Crusaders were pleased, "All we have to do is wait and do nothing", they said. Since they weren't noted for doing much anyway, this wasn't difficult.

However, much to the dismay of the Crusaders, many Novices amongst Berkeley group were promoted to Gurus, and these new Gurus worked to consolidate and strengthen the new rituals. The label of 'slow and unwieldy' was not to be applied again. Time and motion studies were performed on the rituals for the first time in the recorded history of the UNIX cult and a little more than lip service was paid to the notion of efficiency of worship.

### **The Standardisers**

One way of defeating the degeneration of the pure UNIX cult into Sects was by the creation of a 'Standard'. As we have seen, the Svid is one example of this. However, the Svid differed from other Standards because only the AT&T marketeers were in a position to generate a Standard without reference to anyone else. They seemed especially keen that no taint of the Berkeley heresy should appear in their work and so did not consult the Users.

In order that the Svid could qualify as a Standard, the AT&T Marketeers had to promise that it would not alter. They agreed to this because they were determined to ensure that the Svid gained religious significance. This was a good thing for other Vendors who were treating the Svid as a Standard but it is possible that the slavish adherence was detrimental to the fortunes of AT&T in the long run because they were unable to update the rituals and practices to keep up with the demand for change created by the Users.

All the other standards were either created by groups of Vendors working together and ignoring the Users; or by groups of Users working together and ignoring the Vendors. Gurus were rarely involved; they were either too busy and important to sit on committees or just plainly could not see the need for conformity.

The Standards rarely reflected the UNIX practices and rituals which were in use at the time of their creation. All of them seemed to have a speculative element, as if the Standardisers themselves tried to develop or perhaps rationalise the rituals in some way. As a

result, the Standards were never standard.

However, the various Standards did give the Users an idea of what was expected of them if they desired to move from one Sect to another. Users who did this often, the so-called *Portables*, learned to use the minimum of ritual and to localise the Sect dependent areas of their worship.

The Portables might have been helped by a Standard for the religious language, C. They were surprised to learn that Standard C was, in fact, a different language from the original and almost no-one had an Overlord which could understand it. The changes were no doubt desirable but came from treating C as a 'high-level language' rather than using it in the way which the Creators intended, a method of communicating intimately with the Overlords.

### Conclusion

The archives are still being searched for other interesting material but enough has been found to demonstrate the fervent activity which followed the creation of the UNIX cult. We have found no trace of the Creators and barely a hint of the disciples who followed them inside AT&T. We hope that more research may provide some answers and respectfully ask for more funding.

### Biography

Peter Collinson graduated from the University of Essex, UK in 1970. Due to a quirk in the British education system, the University could only grant 'Bachelor of Arts' degrees, so he is the proud possessor of a 'Bachelor of Arts in Computer Science'. He stayed at Essex for three years doing postgraduate work and obtained a Ph.D in 1973.

At this point he moved to the University of Kent to join a growing Computer Science department as a lecturer. UNIX entered his life in 1976, when a colleague was forced to give up the bootstrapping of Version 6 on a PDP11/40. The system soon became an essential part of Kent's computing resources. Several thousands of lines of C flowed and then in 1980, money was found from central government sources to buy a DEC VAX11/780.

The VAX was to have no directly connected terminals but was to talk to the users via a Cambridge Ring local area network. The code to do the networking was originally done on UNIX 32V and then moved to Berkeley 4.0 on the discovery of the delights of the suspend signal and job control. It is running now on six 4.2BSD VAXes on Campus and a machine in New Zealand. On the way, of course, many other programs and systems have been written and like all good code, thrown away when something better comes along.

Peter has always been interested in UNIX User groups of one form or another. He was an early Chairman of the UK user group before the group grew into EUUG, the European UNIX User group. He is a committee member of the EUUG.

## Unix: A Dialectic

Dennis M. Ritchie

AT&T Bell Laboratories

This paper is about why the Unix® system has succeeded, and why it is good, and how its virtues lead to limitations. It is not about what it doesn't try to do, or even about what it tries to do but does badly, but instead is about problems that arise out its very nature and history. I will discuss its computational model, its use of tools, and portability.

### The Model

The Unix kernel embodies a simple, coherent, and powerful model of computation. The operating system itself has only two underlying concepts: the file system, and the process. Once these are understood, the system as a whole is mastered. Moreover, it should be possible to understand them; they were designed to be comprehensible, in the sense that a 9-page paper, published in CACM more than a decade ago [1], still constitutes a description nearly complete enough to serve as an implementation plan.

The file system has two aspects: its static structure, and the operations that can be performed on it. The static structure is specified by the set of file names, and their organization, which is a tree of arbitrary depth. Some files are directories, and contain names of other files and subdirectories. Directories can be read, to enumerate the files they contain.

All ordinary files are simply sequences of bytes, without any system-imposed structure. Text files, in particular, are nothing but characters, with new-line characters to delimit lines. There are no "access methods" or "file types." Of course, programs can write files with any structure they please, but the custom is to avoid complicated formats whenever possible.

The operations are few: create a file, delete a file, read bytes from a file, write some bytes. A newly created file replaces an old one of the same name, and files grow as necessary.

The file system operations generalize to objects other than ordinary disk files. For example, tapes and terminals and network connections behave, to the extent possible, in the same way as disk files. Each is, in its own way, complicated and unique, but a largely successful effort is devoted to making these things behave in the same way as files, so that a program seldom needs to know what kind of object it is talking to. In particular, devices have names in the file system, the same protection mechanism applies to them as to regular files, and the same I/O system calls are used.

The characteristic form of interprocess communication, likewise, is treated like file I/O. The pipe is a connection between two processes, whereby one process writes data that another reads, with buffering and synchronization handled transparently by the system. Of course, the ordinary read and write calls apply to pipes too.

The other fundamental object is the process: a program in execution. New processes are created when an existing process splits, or *forks*. Often the new process immediately performs an *execute* operation—that is, it calls in a new program from a file and causes this program to start running. Processes have only certain kinds of interaction with the rest of the universe. First, when a new program starts executing, it receives several character string arguments via the request that started it. Thereafter, it carries out I/O to files. Some of these it opens or creates by itself, often getting the file names from its arguments. There is a convention by which processes start off with a "standard input" and "standard output," which are pre-opened files that often refer to the terminal but can be redirected elsewhere.

A standard command interpreter, called the shell, works this way: First, it reads a line from its standard input; this line specifies a file containing a command, and some arguments. A new process is created, and the new process calls in the named command, passing it the given arguments. If requested, the shell will adjust the standard input or output of the command to take input from, or place output in, a named file, or it will connect several processes to form a pipeline, in which the output of one command is connected to, and processed by another.

## Tools

The characteristic style of user-level programs exploits the metaphor of the toolbox: the commands constitute a set of software tools, each performing a limited function, that can be combined in powerful and interesting ways. The books of Kernighan and Plauger (see, for example, reference 2) show how the approach can succeed in environments other than Unix. However, the tools work especially well in the world provided by the shell, with its notation for combining programs.

The tool metaphor has led to an explosion of interesting commands. Many of them are small: they are the nuts and bolts of the toolbox. They do things like concatenate files, count words, and search for text patterns. The approach has encouraged new ideas about how to design programs, and how to represent data. The crucial perception is that the output of any program is potentially the input of another. Elementary examples: one asks how many users there are on the machine by sending the output of the *who* command into the command that counts lines; one counts the distinct words in a document by splitting the document into one word per line, sorting, casting out duplicates, and tallying what results.

Larger programs, more akin to power tools, often embody "little languages," and are often special-purpose processors of text. This is most evident in our approach to word-processing. The powerful if rebarbative programs *nroff* and *troff*, developed by Ossanna, provided a way to format documents consisting of ordinary text. The first significant preprocessor for it was *eqn*, by Kernighan and Cherry; it provided a language for describing mathematical equations. Others include *tbl* (Lesk) for setting tables; *pic* (Kernighan) and *ideal* (VanWyck) for incorporating line-graphic material; *refer* (Lesk) for consulting a database of references and incorporating them into the text\*. Most recently, *grap* (Kernighan and Bentley [3]) has appeared; it is used for graphs.

The important observation is that it seems certain that these programs would never have been developed if they had to be incorporated into *troff*. The structure of the operating system, and the type of thinking it induced, encouraged modularity: each tool could be developed independently of the typesetter's internals.

## Portability

A third contribution to the success of Unix is portability. The system is written in C, a reasonably expressive, medium-level language; programs written in it can be moved to a variety of kinds of hardware. The operating system proper makes relatively modest demands on the hardware, and itself is relatively easy to move. Moreover, the system is widely available in source form under remarkably liberal licenses. Consequently, many groups had a chance to contribute to the system, and it was tied neither to the machine on which it was developed, nor to the group that developed it.

People used to think of operating systems as givens, as lumps provided by the manufacturer that they must simply learn, and perhaps petition for changes in. Because of its history as a research effort not a product, this has been much less true for Unix: as it was being developed, people inside AT&T, and in universities, saw it simply as a program that could be understood and changed. Most notably, this apperception occurred within the

\*All of these programs are described in most volumes of the *Unix Programmer's Manual*.

Programmer's Workbench group at Bell Labs (they largely merged into the Unix Support Group or USG, and that group, much larger now, has since become part of AT&T Information Systems), and also at the University of California at Berkeley. There are those who think that the Seventh Edition distribution (or perhaps even the Sixth Edition) from Bell Labs research is the loveliest flowering of the system, but System V and BSD 4.2 or 4.3 are the versions that most people use. Certainly, the contributions from many separate sources were necessary to the system's success.

Portability to variegated hardware was also vital. Unix prospered early because it was first available for a popular machine, the PDP11, and was later moved to another popular machine, the VAX (after portability was demonstrated on a rare one, the Interdata 8/32). It has now propagated to nearly every important machine architecture, from the 8086 to the Cray 2. This development is important, if for no other reason than that availability of a common, understood environment for computing is a valuable good. This value is seen, for example, in the persistence of old languages, such as Fortran. Few, these days, defend the language itself. Numerical analysts, however, are realists: they would prefer to write in other languages, but understand that when an algorithm is expressed in Fortran, it can be communicated to their colleagues, that there are good Fortran compilers on most machines, and that by using it they are increasing their own effort but maximizing their contribution to society. Thus, these three technical characteristics contributed to the success of Unix:

- simplicity and coherence
- the tool-using approach
- portability

Each of these virtues, individually and in combination, has negative aspects. Although I think the negative aspects—the “dark sides,” are considerably outweighed by the virtues, they are fundamental and have to be faced.

### **Simplicity and Coherence?**

Consider “simplicity and coherence.” The first fact is that the system is not so simple as it once was; time and reality have complicated it. For example: most operating systems have an ugly file system, with operations so low-level and complex as to be impossible for application programs to use, and an I/O library to paper over the complexities and ugliness. In part, the file system was designed to make this step unnecessary, so that one could actually understand the operating system interface. Nevertheless, the “standard I/O library” had to be invented for various reasons: buffering for efficiency, and for portability of programs to systems other than Unix. Unfortunately, the I/O library has itself become complicated—more complicated, indeed, than the underlying calls, yet it is the interface that most programs actually use. Why does this matter? For example, the underlying file system can make promises about atomicity of operations. However, these promises are lost when I/O is buffered by the standard library.

The intended simplicity, and concomitant intention that the workings of the system should be fully understood and predictable, leads both to a certain curtness in the documentation and to some blind spots in the services provided. For example, a utility to repair damaged file systems did not appear until relatively late in the development of the system; until *fsck* was written, administrators, at least, not only needed to know the structure of the file system, but how to fix it when it was broken.

More fundamental is this fact: Unix is simple and coherent with respect to a certain model of what you might want from a machine; it makes a strong, visible statement that what you want is an operating system. On these terms, it is excellent. Many people, though, do not want an operating system at all; they simply want the machine to do a particular job. They do not care about files or directories or processes or I/O redirection. These people form the majority of the computer-using population, and may just want to edit and enter text, and get it formatted nicely; or they just want to receive and send mail; or

they just want to create and run their reactor-design or weather-modeling codes. If they are sophisticated, they may see, and begrudge, every machine cycle that goes into overhead.

Many users have no interest whatever in forming a correct model of what is really happening, and using it to predict the behavior of the machine when they try new things. Furthermore, a substantial fraction of them are entirely justified in not bothering even to learn what an operating system is, let alone how to use it creatively. In other words, the underlying simplicity of the concepts underlying the operating system is irrelevant to many of its users, because these concepts are unrelated in any direct way to their desires and needs.

### **Tool Using**

This begins to touch on the second great virtue of Unix: composition of tools. I spoke above of our approach to text-processing. To write a paper with a complicated textual structure, one must deal with a great many programs: an editor, the basic formatter, and some subset of the more specialized tools: one for equations, another for tables, another for graphs, another for references; and there is a whole host of ancillary services available, like spelling and grammar checkers, double-word finders, and so forth. The approach is certainly more forbidding and complex than a single, integrated word-processing program like the ones that can be bought for a microcomputer, because there is more freedom: you get to build things for yourself.

It is easy to argue that the several pipelined programs take more CPU time than they would if they were jammed together. Moreover, there are inherent limitations in pipelined text-processing programs, because the pipeline is one-directional, without feedback. For example, the programs setting tables and equations have no idea where their result will land on the page, which leads to imperfect placement; similarly, the equation-setter suffers from lack of knowledge of the properties of the font it is using.

### **Implications of Portability**

Unix is portable in two senses. First, in the computer-specific way: it can be moved, without great difficulty, to a wide variety of machines. Second, it has been transported to, and through, a great many different groups of people; that is it has had many contributors.

That it had many contributors—the Bell Labs research group where it was invented, the development group that has become a division of AT&T Information Systems, and UC Berkeley—is historical fact. It was born as a research effort, not as a product, and was not described as a whole, even in an informal specification, until recently and it grew over a period of years. It is regrettable, but was probably inevitable, that it split into somewhat differing branches, of which the most notable might be called the System V family and the BSD family.

When development of anything is undertaken by separate organizations, with differing purposes, the products of their development will diverge, as surely as evolution creates new species when similar populations become separated. In retrospect, I can see how certain of the most annoying differences—such as the terminal control specifications—might have been prevented by timely efforts. Other differences between System V and BSD, for example the approach to networking, could not have been prevented, because they developed independently and in different environments. And there is a whole host of small variations in command behavior, and bits of the software like include files, that differ for no important reason; to continue the evolution analogy, they illustrate random genetic drift.

Unfortunately, software portability itself is, for someone who is trying to sell hardware or whole systems, rather an annoyance. To succeed in selling something, a manufacturer needs product differentiation: people should be able to tell the product from the competitor's. This is true even if the selling is noncommercial, as in the academic world. It is valuable to distinguish your ideas from those of others.

So to a manufacturer, there is a tension between providing something unique, and still remaining standard. Some, like Pyramid, have chosen an astonishing tactic: a “dual

universe" machine that will run both System V and BSD programs. Others offer extensions with plausible explanations (or rationalizations) for the departure from the standard—and, of course, if there is no written standard, as there has not been until very recently, the departures are easier to rationalize.

The two-edged nature of software portability is most clearly evident in AT&T's efforts. Our company developed the Unix system, and owns it: it is a proprietary product. However, as history worked out, its market was created by distribution of low-cost software licenses, beginning well before we were selling hardware. Now that we are a hardware supplier, we find that we are, in effect, competing against our own software product.

## Conclusion

To put matters rudely, what I have said is:

Unix is simple and coherent, but it takes a genius (or at any rate, a programmer) to understand and appreciate the simplicity.

The tool-using approach is powerful and intellectually economical, but it takes imagination to use. It may also be more costly to combine simpler, more general tools than to build a more specialized one.

Software portability is socially and, in some ways, economically valuable, and Unix achieves it astonishingly well. But for big systems, perfect portability is practically impossible to realize, mostly for reasons that have nothing to do with differences between hardware.

All these contradictions exist, and should be faced. Rather than attempt a grand synthesis, or make strong suggestions, I will content myself with some small observations.

Although it is unlikely that the differences between the various versions of the system will disappear, the trend does seem more towards convergence than differentiation. Standardization groups like the IEEE P1003 committee should at least expose the issues, and at best may create a politically neutral center towards which Summit, Berkeley and perhaps now Pittsburgh can gravitate. (At worst, they will create yet another variant version.) Moreover, even some of the economically active players in the market seem to be forming technical alliances intended to alleviate variation.

The degree to which Unix can continue to retain its original simplicity and coherence of design is less certain. There is a real struggle between those who would like to make it the operating system for the masses and those who like it the way it is. (A quick test on this issue: how do you feel about "m \*"? Is it too dangerous, or do you accept the logic that makes its meaning inevitable?) There is a related, but distinct struggle between those who find it necessary to add features and those who strive to generalize and extend it in ways consistent with its design.

Most other operating systems are bland; opinions on them run the gamut from "ugh" to "it's OK." At best, they provide a neutral background in which to work, while Unix makes a statement about how to program; it has a style. Correspondingly, it attracts strong fans and vocal critics. It is the only system I know of about which people publish papers arguing that adding features is wrong. It has true believers: people who argue that the more options a command has, the less its author has thought about what it should really do.

The existence of such people is the final endearing thing about Unix, but one that can be annoying, and even dangerously stultifying. It is good, and it is genuinely interesting, to be involved in a system in which aesthetic judgments have such an important influence on the design. It is also unnerving to the populace to be assaulted by fanatics who assert that theirs is the unique road to salvation. One can be trapped by Unixism as much as by any other philosophy.

## References

1. D. M. Ritchie and K. Thompson, *Comm. ACM* 17 7, (July 1974).
2. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading Mass., 1976.
3. J. L. Bentley and B. W. Kernighan, "GRAP—A Language for Typesetting Graphs," *Comm. ACM* 29 8 (August 1986).

## Biography

Dennis M. Ritchie was born in 1941, and received his Bachelor's and advanced degrees from Harvard, where as an undergraduate he concentrated in Physics and as a graduate student in Applied Mathematics. The subject of his doctoral thesis was subrecursive hierarchies of functions. Since joining the technical staff of Bell Laboratories in 1968, he has worked on the design of computer languages and operating systems. After contributing to the Multics project, he joined K. Thompson in the creation of the Unix operating system, and designed and implemented the C language, in which Unix is written. In 1982 he shared the IEEE Emmanuel Piore award with Thompson, and in 1983 he and Thompson won the ACM Turing award. His current research is concerned with the structure of operating systems.

# UNIX: The World View

Mike O'Dell

Maxim Technologies

## 1. Thesis

The word UNIX is now a registered trademark of AT&T, but the *concept* of Unix is not embodied in any particular piece of software. Rather, it is an approach to computing, a way of doing business, in fact, a world view. This view is based upon a small set of simple principles which fundamentally establish the "shape" of the Unix universe. These principles are so compelling they have been borrowed by many systems which came after Unix.

## 2. Simple Principles

Ritchie [Ritchie84] and McIlroy [McIlroy86] have both discussed these principles at some length with a view to how they affected the success of Unix. Our interest, on the other hand, is to examine how these principles have propagated; therefore, rather than belabor the delicate brushwork, we repaint a summary of these principles in very broad strokes:

- (1) A libertarian filesystem does not meddle in one's affairs. First and foremost, it simply saves the bytes presented in conveniently-named containers and faithfully reproduces them upon request. It performs this task without argument or supplications. The filesystem also guards such saved information with a powerful but simple protection model which promotes, rather than hinders the performance of the primary task. This gives rise to a system where files are simple, pleasant, and natural to use, instead of a system saddled with control blocks, buffers, byzantine specifications and file creation operations requiring Papal dispensations.
- (2) Processes are provided as simple, commodity objects which one uses with few reservations as they are cheap and almost as easy to use as an "integer." This gives rise to a world where small, cooperating components can reasonably be used to build systems, and the distinction between "foreground" and "background" processing is one of attention span, not functional capability.
- (3) "Design by omniscience" is a perilous way to build an operating system. This gives rise to open-endedness, rather than all-inclusive featurism. More directly: build composable tools rather than enumerated facilities.

From these simple principles flow the wealth that is Unix.

For this author, the quintessential *Unix* remains the system described in [Ritchie74] some twelve years ago. Unix has survived much progress since then, some real and some imagined, but most of it is, at best, elaboration. In spite of the strengths and weaknesses of the many and varied implementations of Unix, the clarity and transparency of that simple system still show through.

### 3. Influence on Design

Unix has exerted a strong pull on the design of new systems, as will be seen in more detail later. Aside from their philosophical “correctness,” the above principles have profoundly influenced design for several other important reasons:

- (1) The principles are simple and embody an elegant model of computation which is efficiently implementable on a wide variety of machines. This is extended to a kind of *Systemic Minimalism*: “If it ain’t in there, it can’t be broken, wrong, or misengineered.”
- (2) The principles, and by extension, the people who understand them, are portable independent of any possible source code portability. Reusing code is a clear advantage, reusing a brain is an even larger one.
- (3) The principles are relatively independent and so can be imported individually by a new world. The synergism of the principles is inherently limited by isolation, but the validity of the notions central to Unix, even in isolation, has been proven time and time again.
- (4) The principles have established a set of basic expectations for what it takes to do real work. Usually this has been accomplished by Unix tools demonstrating concrete, new techniques for getting the work done easier.
- (5) The principles are open-ended, meaning that the Unix enterprise is free to steal quality ideas and concepts from other computing enterprises, and incorporate the acquisitions into its eternally-evolving framework. Remember Duff’s Law: “When in doubt, steal code.” An interesting corollary is “Never waste time having good ideas when you can steal better ones.”

While Unix clearly derives its strength from these principles, they transcend any particular implementation, Unix or otherwise. Each and every principle has been paid the supreme compliment of being “borrowed” for various new designs. The following three examples show how these principles have been imaged in some of those systems designed after Unix. The degree to which this effect is visible varies greatly, however. One system wishes to be very much like Unix and is quite up-front about it. Another clearly tries to mimic many things while claiming loudly not to do so. The third heads off in a rather different direction, but upon closer examination, simply borrows from Unix in quiet anonymity, while pursuing its own vision of the world. Also, each of these systems has introduced new ideas of its own, some of which Unix has bought back, and some Unix has, as yet, been unable to assimilate. Whether Unix can or should buy back these new ideas, while studiously avoiding avoiding the trap of providing the inclusive-or of all known features, is an interesting question unto itself.

### 4. The Software Tools — A Quick Study

*Software Tools*[Kernighan76] was an attempt to show how it was possible, using the Unix tool paradigm, to improve one’s programming environment even if a FORTRAN compiler and a large, refractory computing beast were all one had at hand. Dennis Hall, Deborah Scherrer, and Joseph Sventek, then of the Lawrence Berkeley Laboratory (LBL), routinely used such computing beasts and they immediately took the book to heart. They implemented everything the book suggested and many more tools, some lifted straight from Unix, some borrowed from other systems. They soon had this “programmer’s toolkit” working on several different machines, learning much along the way about which abstractions “ported” and which ones didn’t.

As explorers and pioneers, they weren’t obliged to retain the historical baggage of “how things have always worked” and thus had the enviable opportunity to “do it right from the beginning.” The command names, command-line interface, library structure, and documentation of the Software Tools are much more regular and consistent than any Unix system, no matter how well preened. For example, the LBL group adopted a generalized, uniform,

command-line parser as “standard equipment” right at the outset and then ensured that everything used it.

A joint meeting with USENIX in 1979 introduced many new people to the Software Tools Project, as it had come to be known by then. Soon, many different groups were busily moving the Tools to all kinds of new systems, thereby discovering the *real* limits of “portability.” Through the Software Tools User’s Group, small task groups were formed to tackle different parts of the portability question. A great deal of work by many people produced an even more portable specification of the “Virtual Operating System” (VOS) interface than was previously available.

Instead of insisting a specific operating system run on each system, the Software Tools Virtual Operating System layer provided a bridge between the local system’s facilities, whatever they might be, and the VOS interface expected by the Tools source code. Sometimes the mappings were straightforward; other times they were quite complex. In several instances, target machines had no real filesystem, just a disk and a few FORTRAN interface routines. In these cases, the Tools implementors provided the first real development environment ever to be run on the machines.

These days, the Software Tools run on over 50 wildly different kinds of computers, as well as atop a similar number of host operating systems. In all cases, the source code of the Tools themselves compiles essentially inviolate; only the VOS layer must be reimplemented (and even this seldom proves to be a “clean sheet of paper” implementation effort). The Tools user interface, moreover, is essentially constant on all of the systems on which it runs. A paper summarizing the experience of the Software Tools Project appeared in [Hall80] (referred to as “the VOS paper”).

How was this amazing level of portability obtained? It was *not* achieved with thousands of `#ifdef`s scattered through the source code. Instead, it stems primarily from picking the correct abstractions for the VOS interface layer. One serious liability of Unix is that for far too long, far too many programs have known far too much about the internal data structures of the system. Fortunately this is changing on some implementations of Unix, but an example is still warranted:

#### Core Dump at the DIRSIZE Corral

When the Computer Systems Research Group at U.C. Berkeley originally announced it was changing the format of directory entries in 4.2BSD and would thereby force programs into using library routines to manipulate directories as opaque objects, the howls and cat-calls were heard far and wide. [How dare they meddle with the Sanctified Fourteen!] The Software Tools community thought all this commotion rather silly. With 50 different host implementations under their belts, they had long since developed a powerful suite of directory and filename manipulation functions. All their programs worked fine, thank you, in spite of the 50 different underlying representations with which they had to contend.

As an example, a single version of “ls” worked everywhere. Admittedly, on some systems, some of the fields in a “—l” report were fabrications, and “ls” was a program implementors were encouraged to customize to some degree. But it worked, right out of the box, on 50 different kinds of computers. Try to say that with a straight face about any Unix program which does a “stat()” system call.

As the LBL VOS paper emphasizes, the power of the Tools environment arises from the synergism of tools working together. Many systems have useful tools, but most often they don’t work together as readily as in the Unix/Tools paradigm. The two key contributors to this synergism are the stream file abstraction and functional composition as embodied in pipes. Along with command line arguments, these would be the most useful abstractions to salvage if one must pick and choose which dragons to slay.

How does one do pipes on 50 different systems, almost all without pipes? Easy: you cheat. The Software Tools shell fakes it with temporary files! The concurrent data-flow

programming model embodied in the shell (Unix or Software Tools) is easily the most powerful abstraction ever routinely placed in the hands of users. Still, the implementation of the abstraction does not require *true* concurrency. Real concurrency is certainly nice to have, for without it, interactively supplying input to a pipeline based on examining its output is impossible. But it is not an *absolute*, steadfast requirement for “usefulness.” The Unix/Tools experience is our best evidence of this. The strength lies in the synergism of functional composition, not particularly in the opportunities to exploit concurrency.

To summarize, the Software Tools Project was the first real offspring of Unix. Many people to this day think it was an attempt to build a “poor-man’s Unix” —a substitute people can use when they can’t have the “real thing.” The Tools were certainly born of profound suffering, but the project became much more than just an effort to bind open wounds. The Tools people learned many lessons from what Unix did both right and wrong, and in the process, wrote many new chapters on portability, defensive programming, and design. By example, they showed that portability goes far beyond the ability to run the same software on different circuit board layouts of the *same* computer (as is the case with CP/M and MS-DOS).

Anyone interested, then, in writing portable software in any language, regardless of the target operating system environment, should learn the lessons of the Software Tools. Note well that these lessons apply most particularly to the C programming language and the variants of the Unix operating system. More Unix programmers should learn these lessons and stop un-inventing the wheel.

## 5. MS-DOS: A Curious Pedigree

MS-DOS is a curious mixture of something old, something new, something borrowed, and something Blue. The something old is CP/M, the RT-11 clonette originally implemented for the Intel 8008 machine. From this legacy, MS-DOS borrowed many of CP/M’s filesystem conventions, such as control-Z characters as EOF indicators (in some types of file) and distinctions between “text” and “binary” files. Also from CP/M came a digraph line terminator, CR-LF (or “crilluff”, as it is sometimes pronounced). Having once fought mightily with this wretched convention, I have on more than one occasion claimed that the single most brilliant design decision in Unix is the choice of a single line termination character. Possibly a tad overstated, but not as much hyperbole as it might first appear.

Rather than quibble about the “something new” that MS-DOS introduced, suffice it to say that the advances include such technological wonders as “abort-and-stay-resident” programs and some astonishingly bizarre memory management schemes. But enough adulation, on with the “something borrowed.”

To be blunt, MS-DOS started with a CP/M foundation and then built on it by borrowing from Unix and the Software Tools environment. A hierarchical filesystem showed up in MS-DOS 2.0. Alas, no links, and horror of horrors, they changed the path separator to the OTHER slash. Still, it was enough like Unix that one could actually use the filesystem to organize things. No longer must the root directory contain the 10,000 random files of the MS-DOS distribution. One could make *directories* to hide them, and then the command-line interpreter could find them with a Search Path. Imagine that!

In the file I/O department, File Control Blocks and Disk Transfer Areas had given way to `open()`, `close()`, `read()`, `write()`, `creat()`, and `ioctl()`. The MS-DOS manual entries for these “new” calls revealed ever-so-thinly-veiled references to their Unix counterparts. And as an even more wonderful gift, the operating system had learned how to keep track of file sizes, down to the *byte*, and give a rational EOF indication instead of fabricating a silly control-Z. It had finally become possible to write a file copy command which didn’t need to be told which kind of file it was copying!

MS-DOS also introduced loadable device drivers which allowed the system to be configured dynamically at boot time. This was a marked improvement over CP/M, where “hacking the BIOS” was a black art unto itself. Examining the documentation for coding a

device driver brought intense *déjà vu* to a Unix internals hacker. There were character device drivers with the expected `open()`, `close()`, `read()`, `write()`, and `ioctl()` entry points. Block device drivers had the usual `open()` and `close()` entry points, and, SURPRISE, the classic `strategy()` routine. The device driver structure, while modified to some degree, was straight out of Unix.

The “load and run a subprogram” primitive in MS-DOS 2.0 was almost exactly like the `spawn()` primitive in the Software Tools. By contrast, the delayed binding provided by the `fork()/exec()` pair in Unix is imminently powerful and elegant, but a real problem to implement in many situations — particularly if the host OS can’t do it. The Tools folks discovered this problem early on and decided (correctly, as time would tell) that a “run <program> as a new thread of control” primitive was a more portable abstraction than the elegant delayed binding model. Sometimes portability does require resisting the urge to use every knob on the front panel.

Along with `spawn()`, the Tools shell implementation of pipes as scratch files appears in MS-DOS 2.0. They even borrowed the Unix term “filters” for programs connected with pipes. (Mere coincidence?) Unlike the Tools shell, however, MS-DOS does I/O redirection in the command-line interpreter since file descriptors, as in Unix, are inheritable (something the Tools can’t count on having in every host system).

Whether the `spawn()` primitive and the shell pipe implementations were consciously borrowed from the Software Tools Project, or whether similar restrictions produce similar solutions is not known for certain, but I do know that several Microsoft people knew a great deal about the Software Tools work. I can’t help but suspect it was a considerable influence.

Other indicators that point to a Unix/Tools influence can be seen in the compilers and their affiliated tools which are available for MS-DOS. Once upon a time, a compiler package barely contained the compiler, its libraries, and little else. A bit later, a compiler package contained the compiler, its libraries, often an improved linker, and maybe even an “editor” of some kind. These days, providing tools like `make` is the real litmus test for distinguishing a programming package meant for writing “real” programs from those intended to simply educate and entertain. The expectations have changed radically, almost completely because of Unix. And it isn’t just that people are demanding the services they’ve grown accustomed to. Programmers sitting down to do real work expect to have real tools because they now know there is no reason for a human to do the sort of work computers can do more easily and conscientiously.

To summarize, the MS-DOS of today is very much a child of Unix, in spite of how loudly marketeers would object to that linkage. The filesystem and process primitives have been borrowed, almost wholesale in many cases. The Unix software engineering approach has dramatically influenced the development tools available on MS-DOS. I predict that as MS-DOS continues to evolve, the distinctions between it and Unix will become smaller, not larger.

## 6. The Macintosh Filesystem — Strange Bedfellows

At first blush, the Macintosh operating system appears to be quite different from Unix. Upon closer examination, however, we discover some of our favorite mechanisms in new and intriguing disguises. We will also discover semantic “loose-ends” which have interesting implications for window-based user interfaces on any system.

The Macintosh OS filesystem comes in two flavors: MFS, the original Macintosh File System, and HFS, the new Hierarchical File System. MFS was a simple, flat filesystem born of the original Macintosh small disks, and distinctly microcomputer view of software. At that time, a microfloppy stored 400K bytes and hard disks, for some unfathomable reason, were considered “rare” (the early prices of Mac hard disks certainly did their part to enforce that rarity), so multi-level structures for organizing files were deemed unneeded for the Macintosh. The volumes were, and still are, separate filesystems with no notion of “mounting” one upon another. The world is a forest of filesystem trees, tied together at the top, creating an

ersatz root, in syntax only. Inside, there are only roots on separate volumes; there is no level which disguises volume boundaries.

HFS was born of pressure from advancing disk technology, not to mention the users of that technology. Soon, 800Kbyte microfloppies appeared, and along with hard disks that were getting bigger, faster, and much more common. Simply put, the old MFS algorithms simply didn't scale. The Macintosh needed a mechanism for organizing files beyond simple disk partitioning. What the Macintosh needed was Directories! The HFS filesystem provides directories for organizing files, but the HFS implementation of directories is rather different from the Unix implementation. In HFS, filename paths are stored in a tree structure disjoint from regular files, while in Unix, the filename tree is embedded within the filesystem using directory files as nodes in the tree. Additionally, Macintosh files are not reference-counted as in Unix.

While the implementations are quite different, the motivations behind HFS and the operations on it are very similar to those of the Unix directory structure. Changes in the Unix World View, such as the advent of directories as opaque objects, made it easier to see such functional parallels, even on the face of significantly different syntax. We continue our search for threads of influence by examining Macintosh files.

In the Macintosh OS, files are composite objects, or more simply, you get two for the price of one. Files have two components: a *data fork*, and a *resource fork*. Fundamentally, each fork is the familiar byte stream file popularized by Unix: they are structureless, randomly positionable, and record lengths accurately. They are even serviced by a buffer cache in the operating system proper, but there are some interesting differences.

For the data fork, the Macintosh environment uses a line termination convention different from Unix: the Carriage Return instead of the Line Feed. But to their credit, (and my eternal gratitude) the Macintosh OS designers only picked ONE! So, while the convention is annoyingly different, it is not crippling so.

The resource fork of a Macintosh file is much more novel. Simply, resources are named, structured objects recognized and managed by the operating system. Folklore claims resources were originally invented to circumvent the lack of initializations in the programming language Pascal, but their use became pervasive. Resources are now used to provide a simple virtual memory mechanism for the Macintosh. A component of the Macintosh operating system, called the "Resource Manager," rides herd on resources, loading them from the resource fork into memory when needed, managing memory to contain them, and generally keeping track of them, whether in memory or in a file. It performs this task using a sophisticated index structured stored in the resource fork along with the data of the resources themselves. The Resource Manager, then, is just another client of the simple, Unix-style byte-stream facilities provided by the filesystem proper. While the operating system uses resources extensively, the Resource Manager needs essentially no more support from the filesystem than any other filesystem client. The only special capability provided is the mechanism for binding two byte-stream files, one data fork and one resource fork, under one filename. When one innocently opens the file name, one reads the data fork. Manipulating the resource fork is the province of the Resource Manager system calls.

The Macintosh filesystem is a wonderful example of how an operating system can provide an extremely sophisticated symbolically-indexed "access method" for files while keeping the underlying filesystem based directly on the Unix model. Maybe this example will give pause to some of the more violent claims about what simple base filesystems can and cannot do.

Regrettably, however, not everything is sweetness and light in Macintosh land. Two very subtle operating system design decisions have wide-ranging implications for programs in the Macintosh world, and it is quite instructive to examine them.

### 6.1. Two Interesting Complications

File with two forks are a radical change. No longer can one write a simple file copy program —files are no longer simple! This has serious implications for Tool-based philosophies in this environment and the resolution is currently quite unclear. On the other hand, the resource fork addition makes some things much simpler. Often one wishes to store some attribute information “out of band” but clearly associated with some file. Some systems provide lots of file attributes built-in; some even provide uncommitted bit-fields in file headers. But the generality of the resource fork is compelling. Many different objects of essentially arbitrary sizes and shapes can be associated with the data in a file. Storing font information, “style-sheet” document layout information, or record layout descriptions for self-describing data files are trivial given the Macintosh resource fork. A most intriguing question is whether the power of this facility can be “harmonized” some way with the Unix/Tools philosophy. Is this something worth trying to buy back?

A much more interesting problem is that the Macintosh user interface seems to have no notion of “current directory.” This ties in very subtle ways with the semantics of the filesystem, in fact, because with HFS, the system really has three notions of “current directory.” This is the root of considerable confusion.

Because of required compatibility with programs written for the old MFS interface, the HFS must be “forward compatible” in some fashion. It accomplishes this by providing a “search path” for open()s of existing files (as opposed to creat() operations). In this case, HFS looks in three directories (“folders”) —the “document folder”, the “application folder”, and the “blessed folder”. Translating from the Macineese, documents are data files, applications are executables, and blessed is the home of the kernel image (the “system files”). So, when one tries to open an existing file, as in an editor, the system searches (1) the directory where the previous document was found, (2) the directory where the program itself was found, and (3) the directory where “/unix” lives. Much of this is a hold-over from how people organized information given small, flat filesystems on floppy disks, but it is important none the less, because at least in the case of existing files, this heuristic usually does exactly what you want. The biggest complaint is that the search path isn’t changeable. When creating files, however, users can have real problems.

In the Macintosh shell, called the “Finder”, one normally runs a program by “opening a document”. When the user selects “Open” from the File menu, or “double-clicks” a document with the mouse, the Finder automatically locates the program appropriate for opening the document and launches it with the selected document filename as an argument. Before actually doing the launch, the Finder sets the notion of “document folder” to the folder containing the document. This information is then conveyed to the application program through the OS-provided facilities. For a program not launched by opening a document, but rather by opening the application directly, there is no notion of “document folder.” The only directory the program knows about is the *application folder*! This means that the first save of a new file must establish the desired directory for the file via navigation from the ersatz root of the filesystem, or it will be saved in the equivalent of “/bin”! Why is this? The user interface doesn’t really have the notion of “change directory” —one can see several places at once, so there isn’t the strong Unix notion of “here” (as shown by pwd). There is no way in the current Finder to cruise over to the desired destination folder for a new document and launch an application, without opening an existing document, and get that folder as the default document directory in the application.

Window environments on Unix also show signs of this problem. An iconic interface where one can zip around the filesystem simply by moving the cursor to another window is both valuable and problematic. The ability to maintain separate, distinct state per window is, of course, necessary and desirable, but some ability to “slave” windows together so they track some common notion of “here” is also needed. Navigation in the Unix filesystem has never seemed difficult when there was only one place to be. But with a window system, the user is not really “in” the filesystem, but potentially above it, looking several directions at one.

Firesign Theatre said it best: "How can you be in two places at once, when you're not anywhere at all?" [Firesign71]

The Macintosh environment was influenced by Unix notions of files and directories, created some interesting new mechanisms atop them, and performed some interesting experiments altering the fundamental shape of the world in novel ways. It is valuable for the Unix community to examine this work and the results. There are some things we may want to borrow back, and some mistakes we probably don't want to repeat. If we don't look and learn, we will never know.

## 7. Troublesome Unix Semantics

The semantics of Unix are characterized by simplicity, economy, and an astonishing lack of boundary conditions. In Unix, the boundary conditions are largely finessed by reference-counting objects in favor of sharing and retention. This decision certainly produced a system with graceful curves instead of sharp edges, but the resulting semantics are the bane of some new implementation efforts trying to achieve "Unix semantics" on new platforms, i.e., in distributed environments or atop other operating system kernels. Retained files and shared file descriptors are classic examples of such "troublesome" Unix semantics. These new implementation efforts are raising questions as to the value of some of these mechanisms versus the cost of their distributed implementation. The answers aren't easy, but the issues are very real, and people are making decisions about what facilities will be available in future systems. The community had best pay attention to these matters.

### 7.1. Gone, But Not Forgotten

Retained files arise because in Unix one can `unlink()` an open file. This produces a file which has no name; it remains known to any processes currently having it open, and it can be inherited across `fork()`s and `exec()`s via open file descriptors, but it cannot be opened by another process. Furthermore, when the last process referencing the file `exit()`s, the file's reference-count will be reduced to zero and the file will disappear. The classic use for retained files is as temporary files which do not have to be explicitly removed when a program is finished with them. They simply evaporate upon `exit()`. This has the distinct advantage of not producing trash in the `/tmp` directory when a program is unexpectedly terminated by user action or "natural causes."

This beautifully-rounded boundary condition caused considerable consternation for the designers of one particular telekinetic filesystem which prides itself on using completely stateless servers. The notion of "gone but not forgotten" is particularly difficult to capture when the fileserver has no notion of `open()`s in the first place.

The first solution to the problem was to simply wave it away with one sequin-gloved hand, pronouncing it "a disgusting Unix-ism." The would-be purist immediately discovered just how many programs believed in the smooth continuity of the boundary he had just creased. They were dropping like flies!

In mild defense of the approach taken, there are better ways to package access to temporary files. These engineering approaches generally require the admission that temporary files are somehow different in some subtle ways from other data files. That admission is already in place to some degree: temporary files are already placed in special directories, and even some Unix programmers use functions to generate names for them. (This is another area where the Software Tools are miles ahead.) Admittedly, while it is bad engineering to expect retained files as the implementation of temporary files, it is completely reasonable for Unix implementations of temporary files to use retained files. Therefore, they had better work correctly, or we must rethink this semantic boundary condition.

The offered solution to the fileserver retained file problem was to migrate the expected semantics back into the client kernel. When a file on a fileserver is unlinked, the kernel checks its internal state to detect whether the file is currently open. If so, it postpones the `unlink()` and merely disguises the target filename until the final close, at which point, the

disguised file is actually removed. There remain at least two problems. The first is fairly obvious: if the fileserver client crashes before removing logically unlinked but not closed files, junk can be left lying about. Periodic daemons can readily address that problem. A much more subtle problem occurs when the kernels of two different fileserver clients have differing views of a file's current state. Assume a process on one client has the file open, and a process on a different client tries to unlink() it. The kernel of the unlinking client will not postpone the actual unlink operation since that kernel is unaware of any other interest in the file. The file will then instantly evaporate, leaving the process still using the file a rather nasty surprise!

One can certainly argue this might not be an unreasonable semantic definition for "removing an open file," if it weren't for the well-established retention semantics. One can even argue that programs which interact without cooperating more explicitly are poorly designed. Again, the opposition would be on thin ice except that such explicit synchronization has always been unnecessary because it was provided by the operations, and that naturally synchronizing such operations makes them so much cleaner and easier to use. The architects of the fileserver claim the current scheme is a workable solution and that this last odd scenario doesn't really come up in practical use. While it is indeed workable, it is somehow not completely satisfying.

About this time, some readers will be diagnosing the problem as one of designs using stateless servers, rather than servers with more intimate knowledge of what is really going on. I am not an absolutist when arguing for stateless servers, but to be taken seriously, any alternative fileserver architecture must exhibit the robustness currently available from the stateless designs. Clients should expect to survive server outages with no more inconvenience than simply delayed progress. The first time one witnesses a large server crash and recover transparently, without any loss of work, one becomes a real believer in robustness, if not statelessness.

## 7.2. Let's Take Turns

Shared file descriptors are a shining example of how one simple mechanism obviates much complexity. When the output of a shell script is redirected, the output of each command in the script is naturally merged into the correct order without any special mechanisms. Unix users think nothing of this. "How else would it work?" Accomplishing the same thing on many other operating systems ranges from "impossible" to "extremely messy", requiring special arrangements and global environment flags indicating whether a program should "append" to what we call standard output or should simply "write" to it. This behavior in Unix arises simply and naturally because the file descriptor for standard output is shared between the parent shell and all the children run in the course of the script. Part of the state information shared via the file descriptor is the current byte offset in the file. Therefore, when one process finishes, the offset simply stops moving. When the next process inherits the file descriptor, further write() operations simply move the offset forward from the current position. Not doing an open() operation is crucial to this process, since an open() on almost every operating system initializes the file offset to some value (usually zero!), thereby making it difficult to inherit the state from the previous process.

Several designers seeking to implement "Unix semantics" atop different substrate kernels have discovered this problem. Systems attempting to provide Unix semantics atop message-passing kernels, as well as two commercial "Unix atop VMS" products have addressed this problem with varying degrees of success.

Many message-passing designs, as well as some traditional systems like VMS, treat files as the level of virtualization and sharing. Unless this subtlety of Unix behavior is specifically considered in the filesystem interface design, state is usually maintained per-file, and per-client-process, but with no provisions for client-client coupling. The introduction of "stream state" as an object which can be shared in subtle and fine-grained ways, all intimately related to the process family tree, has dramatically complicated several designs.

One commercial Unix-on-VMS product simply fakes it for the special case of shell files. This solution is less than wonderful. A different Unix-on-VMS system, Unity from HCR Computing, goes to great pains to emulate the Unix environment as exactly as possible. This requires several support processes beyond the actual running Unix program just to manage shared objects like file descriptors. This results in serious interprocess communication traffic which is rather costly. The implementors of this product decided that faithfulness to the semantics was worth the performance penalty necessary to get it. Note that the Unity implementation is a single-processor solution capable of exploiting shared memory. Distributed solutions can be even more costly if not considered very carefully.

The designers of the CNET Chorus system [Armand85] use two Chorus processes to emulate one Unix process, in addition to the filing agents implementing the filesystem and file interface proper. Much like the Unity system, one shadow process is responsible for coordinating the control of shared information, while the file descriptor sharing seems to be done in the filing agents themselves. The document cited isn't completely clear on this point. The Chorus filing agents seem to have been designed with an intimate knowledge of the Unix semantics and with the intention of supporting them. Based on other random, oblique comments, it appears that designers moving from other systems to Unix are often quite surprised to learn what was necessary to support "Unix semantics."

Unix semantics are wonderful and subtle, but the boundary conditions seem hard to distribute. Can one build a truly distributed Unix which isn't saddled with these problems? Should we even want to build one if systems based on distributed services can accomplish "enough" of the same goals? Several groups are trying very hard to build distributed systems; some are even claiming preliminary success. As of this writing, real, working results with real, competitive performance are rare.

## 8. Conclusion

Unix is indeed software; that's why we use it! More importantly, it is also an attitude, an approach to computing, possibly even a state of mind. This paper has explored how this Unix world view has shaped some of the systems that came after Unix, and how Unix will inevitably be shaped by them in return. The power, simplicity, generality, and non-interference of Unix have been exported with great success. It remains true, however, that Unix still has much to offer the next generation of systems, and can still learn a few new tricks in the process, even if you believe totally distributed everything and transparent network widgetronification running on 10\*\*14 MIPS processors are the foundations of The Real Future.

## 9. Acknowledgements

I would like to thank the Program Committee, Uncle Neil, Aunt Beth, Brother Max, and especially Cousin Mark for their considerable help and assistance.

## 10. Biography

Michael D. O'Dell received his BS and MS degrees in Computer Science from the University of Oklahoma. For several years he was Coordinator of the Engineering Computer Network at the University, where he guided the acquisition and installation of the first independent computer system on the University campus. After receiving his degrees, he worked for Lawrence Berkeley Laboratory where he conducted research in Department of Energy projects, with an emphasis on network-related issues; concurrently, for a brief period, he and M. Karels were the principle programmers on the 4.2 project at UCB. Mike left LBL to become Senior Computer Scientist and Director of R&D at Group L Corporation, a startup company building software and systems products for the Information Industry. His current position is Computer Scientist for Maxim Technologies.

Mike has been an active and vocal proponent of the UNIX operating system and world-view virtually since its inception. He has been a key figure in the USENIX organization and has played a leading role in the evolution of the UNIX community as well.

## 11. Bibliography

- Armand85. F. Armand, B. Deslandes, M. Gien, M. Guillemont, and P. Leonard, "Towards a Distributed Unix System: The CHORUS Approach," TEC I.001.1 (Draft), CNET/INRIA (September 1985).
- Firesign71. Theatre Firesign, *How Can You Be in Two Places at Once, When You're Not Anywhere at All?*, Columbia Records, CS9884 (Circa 1971).
- Hall80. D. Hall, D. K. Scherrer, and J. Sventek, "A Virtual Operating System," *Communications of the ACM* 23(9) (September 1980).
- Kernighan76. B. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, Massachusetts (1976).
- McIlroy86. D. M. McIlroy, "The UNIX Success Story," *UNIX REVIEW*, pp. 32-42 (October 1986).
- Ritchie74. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* 17(7), pp. 365-375 (July 1974).
- Ritchie84. D. M. Ritchie, "The Evolution of the UNIX Timesharing System," *AT&T Bell Laboratories Technical Journal* 63(8 Part 2), pp. 1577-1594 (October 1984).

# MANAGING THE DEVELOPMENT OF PERFORMANCE-CONSTRAINED UNIX-BASED SOFTWARE ON MICROCOMPUTERS

Lawrence B. Perkins

Martin Marietta Corporation  
Information & Communications Systems  
P. O. Box 179, Denver, CO 80201

## ABSTRACT

Operating systems which provide superior software development environments are not generally optimized for high performance of the subsequent executable code. This is aggravated in real-time and other "performance-constrained" systems, such as distributed-function terminals. UNIX is a particularly vulnerable example, but it has become the O/S of choice for a wide variety of sophisticated workstations. I shall demonstrate that this offers a management challenge, rather than an insurmountable technical deficiency with UNIX, and shall provide examples to support the position.

## INTRODUCTION

To begin at the beginning, a technically ambitious terminal subsystem was recently designed and implemented. It met all of its required functional objectives: it provided a complex but user-friendly electronic desktop environment, somewhat like that found on a Macintosh, tailored to a specialized user application; it furnished the X.25 Local Area Network (LAN) interface to a host mainframe-resident data base; and it replaced a previous "legacy" system based upon 10-year old "dumb terminal" technology, with which users had become thoroughly adept (and upon which their expectations for the future were honed.) During integration and final testing, however, the new subsystem exhibited a performance characteristic impossible to correct: its interactive response time was too slow to be usable in actual operations.

Much management consternation ensued, and some hard questions were asked. How could the metric performance requirements have been so grossly overlooked? How could the software, designed to provide only a man-machine interface, with no real "number-crunching" functions, have such incomprehensibly high overhead? (For example, a relatively straightforward function, building and displaying a bit-mapped window, took about 5 minutes.) Finally, what could be done to cure the problem?

As the answers began to emerge, it became clear that the solution to this performance dilemma - the cure - would entail an extensive architectural overhaul. And the favorite culprit at the time was UNIX. "Oh, it's UNIX's fault," everyone liked to say. "Boy, you shoulda known UNIX is a terrible performer." Et cetera. In fact, as one might suspect, the fault lay in a number of design elements and decisions, all amenable to proper technical management. The next few paragraphs describe what

happened to create this situation.

## THE USER INTERFACE

For convenience, I shall call this subsystem - which is part of a larger classified system - the Integrated Display Subsystem, or IDS. The IDS user is a photographic analyst, who works interactively with the terminal to fetch background information from a dynamic data base on the host computer, and to periodically update that data and produce reports. The IDS is thus the principal clerical tool available to the analyst, and is in fact the only instrument with which to interact in this way, there being no verbal backup method. The most important difference in the analyst's interface, between the legacy system and the new IDS, is the multi-window electronic desktop offered by the latter. Using such features as "SunTools" and "SunWindows," the IDS is required to provide an environment of 10 active windows, each containing up to 8 panes or sub-windows, at any one time on each workstation. (In practice, it is expected that an analyst will activate only about 5 windows of 5 or 6 panes each.)

The format and content of the windows and panes is controlled by a Host/IDS application layer protocol, in which formatting templates or "forms" are transmitted to the terminal, followed by data to populate the screen in the prescribed arrangement. Similarly, data base updates and window requests are transmitted from the terminal to the host with accompanying forms to define them. Each analyst's access rights and session status are maintained by host-resident "task-manager" software and control tables.

## THE HARDWARE

The entire IDS comprises about 850 terminals, based upon the Sun Microsystems Model 2/120 and 3/160 workstations. (The design limit is 1000 terminals.) The workstations are deployed in "clusters" of from 2 to 4 nodes, Ethernetted together and gatewayed into a proprietary LAN supplied by another contractor. There are about 280 such clusters, and Figure 1 illustrates the hardware configuration of a representative 4-node cluster. Key performance-related items are:

- o Limited main memories, the clients having only enough space for the UNIX kernel and a small application process or two;
- o Shared disk on the file server, to provide all disk I/O operations;
- o Lack of a removable external storage medium (tape or disk);
- o Narrow bandwidth of the LAN gateway (BIU);
- o The Motorola 68010, operated in virtual memory mode, with only 8 processes mappable for execution at any one time.

The configuration was chosen with the combined considerations of effective resource sharing and cost. The latter rises dramatically, of course, with the addition of

components, due to the multiplicative factor of a large terminal population. Note that the 10 megabit Ethernet, in this mini-LAN, has not presented performance problems.

## THE INITIAL SOFTWARE

The baseline operating system, for both software development and workstation operations, was Sun's Release 1.2 of Berkeley UNIX 4.2, with Sun enhancements. Applications software was developed entirely in C, and comprised about 80,000 lines of code and embedded data tables. It was decomposed into 30 functional processes and over 1800 individual modules. Figure 2 depicts the process architecture of the software. With the exception of the COMX25 process, which inhabits only the communications server, a copy of this code runs (or attempts to run) on each node in a cluster. Key performance-related items in this configuration, are:

- o The software text, not including the UNIX kernel, is about 4.6 MB at load time. Figure 3 illustrates the memory map at this starting point. As the code executes, however, dynamic growth results from the profligate use of the system calls "fork/exec" and "malloc." Ultimately, many more than 30 processes are competing for resources, and the required swap space to support them approaches a maximum of 55 MB. (In practice, the virtual to real ratio - V:R - is artificially limited to 18:1 to avoid system crashes.) Thrashing sets in rapidly, CPU absorption approaches 100%, the paging rate exceeds 20 per second and no useful work is done for long periods.

- o Interprocess communication relies heavily on file passing, using UNIX File I/O. For example, tests have shown that opening and immediately closing a window, with no useful display of data, requires 85 file opens and closes, and 2000+ disk reads and writes.

- o The file server must supply all disk support for the cluster, including paging service, applications file passing, and file service for the communications packages in all nodes.

- o The UNIX dispatcher is a time-sliced, round-robin task scheduler. Its priority adjustment capability is limited and, in any case, cannot be manipulated to tune the response to externally-generated asynchronous interrupts.

- o The UNIX file scheduler optimizes for the predicted position of the disk arm, and builds its queues accordingly. What is thus quite appropriate for a time-sharing application is inadequate for a real-time, disk I/O-intensive load of file passing and heavy paging. Further, the file server itself doubles as an IDS workstation.

- o The LAN implementation uses a full ISO OSI 7-layer communications protocol, with the attendant overhead of embedded conversation control and required processing, which approximately halves the effective data bandwidth.

## THE DESIGNERS

I would be remiss not to offer some defense of the IDS software design

team. Several influences contributed to their inability to control performance during subsystem development, not the least of which was a limited level of experience with UNIX and C. Other driving factors were: recovery from a one-year schedule slip induced by the default of a previous contractor for (non-UNIX based) workstation hardware and system software; conversion of an in-place applications design based on that contractor's architecture; severe shortage of UNIX systems programming (internals) skills; programmatic requirements to produce a full top-down, hierarchically decomposed, modular design, with formal reviews, testing and Milstad documentation; and reduced visibility into other system components because of too few appropriate security clearances. This team probably did as well as could be expected, given the targets and direction they received.

## THE CASE AGAINST UNIX

The management techniques first used to circumvent this early crisis situation might be called "divide and conquer" and "fix it later." About 26 people were split into small groups along software functional lines, and directed to build a diverse set of independent UNIX processes, tied together by file interfaces. No performance restrictions were established. Indeed, performance was expected to suffer, through not to the extent that later developed, and it was theorized that a subsequent remedial effort could bring it to acceptable levels. A better understanding of UNIX, in retrospect, would have surely diverted the group from this path. It is clear that any high-performance application must be carefully tailored, from the outset, in its run-time use of UNIX system services. It must never be allowed to mimic a development environment.

It is not at first obvious, from a management viewpoint, that this is so dramatically the case. As many of you know, it is almost universally true that off-the-shelf (OTS) operating systems which provide superior development environments are not optimized for high performance of the subsequent executable code. This dichotomy is aggravated for real-time, near-real-time, and what I have called "performance-constrained" subsystems such as distributed-function terminals. But it is often possible to obviate the worst offenders in a given system through proper programmer training and judicious selection of services. The usual alternative, of course, is to implement a real-time executive for the production system.

UNIX is a particularly vulnerable OTS example, however, if one proceeds without specific design guidelines and restraints in the use of its very powerful features. This assertion is especially applicable to current microcomputer systems, with their relatively limited storage and bus-type I/O architectures. That UNIX has become the O/S of choice for a wide spectrum of sophisticated workstations (Sun, Apollo, and AT&T products exemplify this; and we have not found a compatible real-time exec for the Sun) demands that software managers be prepared to deal with some new development approaches. Further, the ever-widening symbiosis between UNIX and programmers, especially those emerging from university computer science curricula, demands that we recognize and cope with a new personnel administration phenomenon. To many of these programmers, anything less than a UNIX environment is considered practically paralyzing!

This presents a management challenge that extends from the very early phases of development (personnel, software and hardware selection; training; basic design) through detailed design (coding and debugging) to final acceptance (formal testing and turnover) of the production software. Most of all, since some supervisors may be a bit on the "vintage" end of the calendar, it implies a need for some fairly extensive management education in the UNIX idiom.

Some may argue that one man's "new" challenge is another's obsolete one; that advances in the technological state of the art have overcome capacity and resource limitations. I concede the point, except for performance constrained systems. Then one must go back to basics. A millisecond is still a millisecond, and an event that takes 200 of them is easily discernible to an interactive terminal operator. Furthermore, that operator's "state of the art" has also advanced. Response times, which were once artificially regulated to the 2-5 second realm, have given way to sub-second ones which cause the modern experienced operator no discomfort whatever. We must therefore impose the appropriate limitations on the design process for such systems, if we are to have products that are superior in the eyes of that ultimate critic, the end user.

## THE CASE FOR UNIX

Well then, did the IDS designers use the the wrong O/S? The evidence from my second example will indicate otherwise. Once a software management staff has assimilated the basic strategies and design precepts of UNIX and its user community, they should be prepared to adapt its capabilities to almost any project. There are no compelling reasons for abandoning this O/S altogether, especially when fully supported OTS versions are so prevalent; and the alternative may be prohibitively expensive in time, money, and personnel commitment. The key word is "adaptation", and it applies equally to UNIX and the organization using it. Let's explore an example of this adaptation.

## THE "FAST SOFTWARE" CURE

A hand-picked team of seven people was assembled and tasked with developing a proof-of-concept "Fast IDS". UNIX was the unequivocal choice, since by then time was too precious to consider a real-time exec approach. No earlier applications software was deemed salvageable, due to its embedded architectural commitment to expensive UNIX features, so a rewrite was dictated. Divine inspiration did not intervene in our behalf, but some well-tested tenets of real-time system design did:

- o For instance, it is well-understood that all critical components of a real-time system must be resident in real memory, immediately available for execution, at least with today's memory and mass storage technologies.
- o The virtual to real (resident) space ratios for typical successful systems range between 2:1 and 5:1 for batch operations, and remain at about 1:1 for performance constrained applications.

- o Most virtual memory functions (region management, page aging algorithms, swapping control, and the associated I/O load) are typically relegated to system software. The currently available hardware-assist becomes insignificant, especially as swap rates approach the physical limitations of the external storage devices.
- o It is imperative that the task execution priority of a real-time system be tunable at test time, and preservable in its test-proven configuration at operational run time.
- o In general, small things, with the possible exception of "DO FOREVER" loops, run faster than big things.
- o Many, many otherwise competent (possibly superior) programmers do not know these things, even though they may have experienced their effects, perhaps even in non-real-time. They will respond to expert guidance.

The Fast IDS team members were selected for their specialized skills; one belonged to the original design group, one was an incurable hacker, one had Host computer experience, two had some UNIX internals exposure, all knew C and, to round out the group, a full-time Sun systems programmer was added, with enthusiastic support from his company. All were volunteers who, in spite of their colleagues' opinions, felt that the necessary performance was achievable and had ideas of how to get it. Once properly indoctrinated, the team had no dogmatic fixations about what was permissible in the architectural or implementation approach. (Translation: structured design was allowed, but not mandated.)

## NEW DESIGN GUIDELINES

Taking a page from the airborne computer software book - an environment in which system resources are typically limited and cannot be expanded - we adopted a budget management approach. Limits known to affect performance were set in advance and monitored throughout development:

- o Source code not to exceed 30 K LOC (excl. of COM);
- o Loadable text not to exceed 2 MB (1 MB goal);
- o V:R ratio less than or equal to 1:1;
- o Processes less than or equal to 10 (fork/exec forbidden);
- o Use pixrect, rather than SunTools/SunWindows;
- o Use sockets, rather than file I/O;
- o Assembler permitted if needed;
- o Response time (local) 2 - 5 seconds (1 sec. goal);
- o Trade space to get speed, but prudently;
- o The users' needs are dominant;
- o If it works, don't fix it.

The resulting subsystem, which has advanced from a proof-of-concept to a replacement for the original, is shown diagrammatically in Figure 4. A total of 6 processes embodies all of the functionality, with only 5 of these active after Startup has initialized the system. The COMX25 process runs only in the COM Server, so all other nodes must accommodate only 4 processes during normal operations.

(Refer to Figure 6 for a recent update. The X25 protocol handler has now been rewritten and moved to the Programmable Line Interface Module, where it executes on an 80286 CPU. This relieves the COM server of much interrupt intensive low level processing in the network layer.)

Figure 5 presents a memory map of the Fast IDS. Sizes are approximately in accord with the guidelines; 38 KLOC source and 2.5 MB text, the resident set size being about 900 KB. The kernel is intact, at about 800 KB (recently stripped to about 480 K), and swapping is insignificant. Speeds vary from subsecond, for simple functions (pop-up menus), to 7 seconds for complex ones (initial window builds). This product is not "top down" and it contains some unconventional constructs, but it works, meets user requirements, and is very fast.

## CONCLUSIONS

Perhaps the best way to draw conclusions is to explore what was done differently in the second approach. First, the small number of highly-skilled specialists was carefully chosen, co-located in two adjacent rooms, given all the hardware resources they required, inculcated with the notion of team action and team success, and constantly reinforced with the importance of the performance element in their work. You would like to have them in your shop!

The strengths and limitations of UNIX were analyzed and selectively exploited. We went to great lengths to differentiate between the appropriate development and production uses of system services. High-risk plans to modify the kernel were thus reversed, in advance.

The fundamentals of real-time resource management, were practiced during development and testing. Incremental consumption of various capacity parameters (memory, swap space, swapping rates, CPU utilization, etc.) were regularly tracked and evaluated, thus avoiding nasty surprises late in the game.

Conservative estimates of development time, albeit in a relatively informal mode, were used from the start. The project was able to stay virtually on-schedule and avoid the sometimes destructive management pressures induced by lateness. The rewrite took 10 months, including the implementation of a hierarchical file system to replace file I/O and an integrated word processor window to replace the vi editor used in the initial system.

UNIX has proven itself to be an adequate O/S, when used in this controlled fashion. (As the functionality of the terminal grows, however, a point will come where a real-time exec trade study will definitely be in order.)

I believe we have demonstrated that attentive management of the whole development process, with proper emphasis on the power - and potential for degradation - of the UNIX environment, can produce an effective, responsive, performance-constrained system on a large population of small machines. I can think of no reason that these management principles are any less applicable for other

Unix-based application developments, on other classes of computer hardware. After all, it is on the computer itself that theory meets reality, and until the technology allows us to have a virtual world, reality prevails. There are still resource constraints, and we must still manage them.

#### References:

T. Ferrin, "Beating The Real-Time Rap", UNIX Review, Vol. 4, No. 4, pp. 36-46, April, 1986.

D. Chappell & D. Welch, "Implementing OSI Protocols In The UNIX Environment", Systems & Software, pp. 73-76, November, 1985.

L. Perkins, "Managing Performance in the Real World", Fifth Annual Conference, European Computer Measurement Association (ECOMA), London, England, April, 1978.

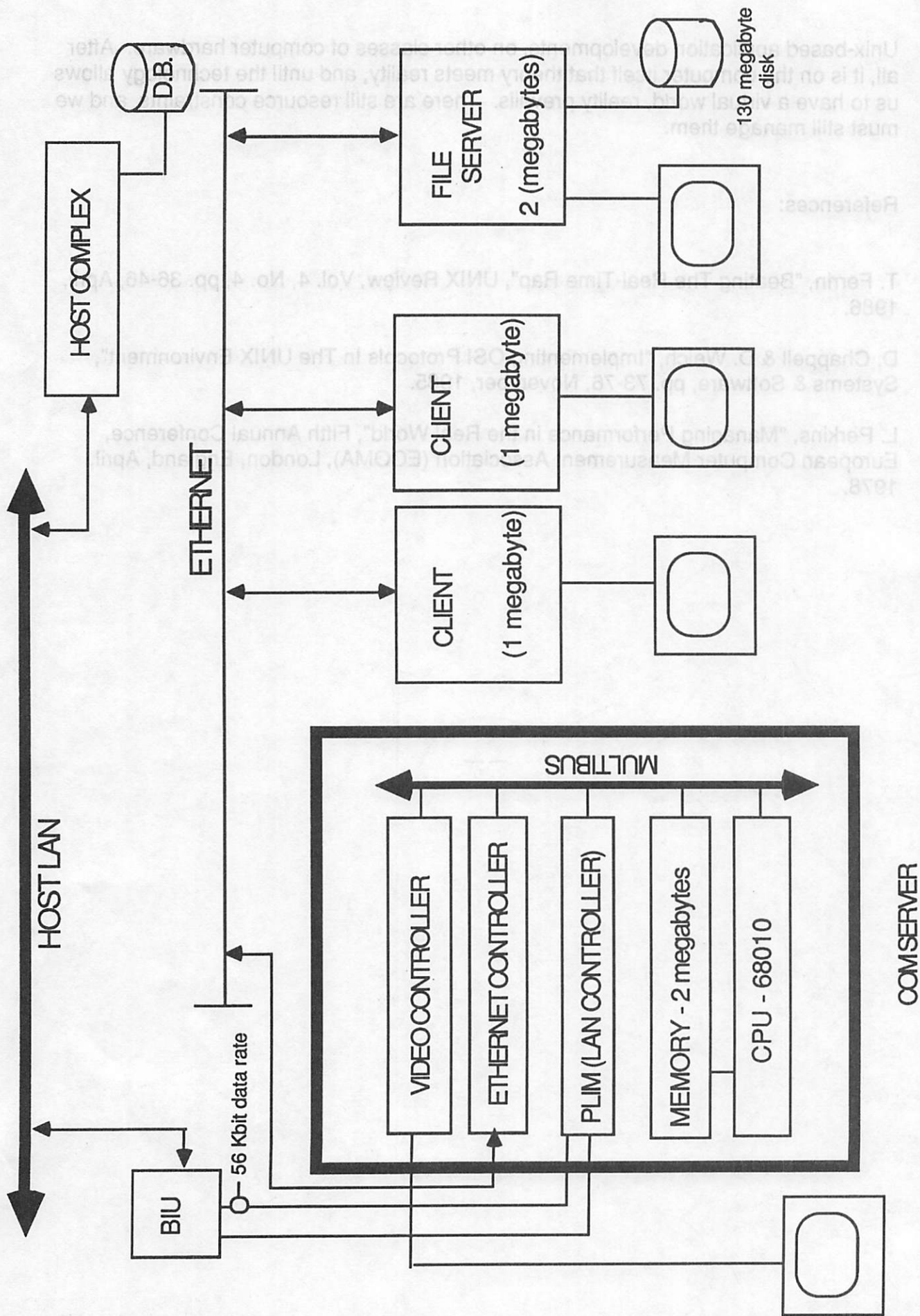


FIGURE 1

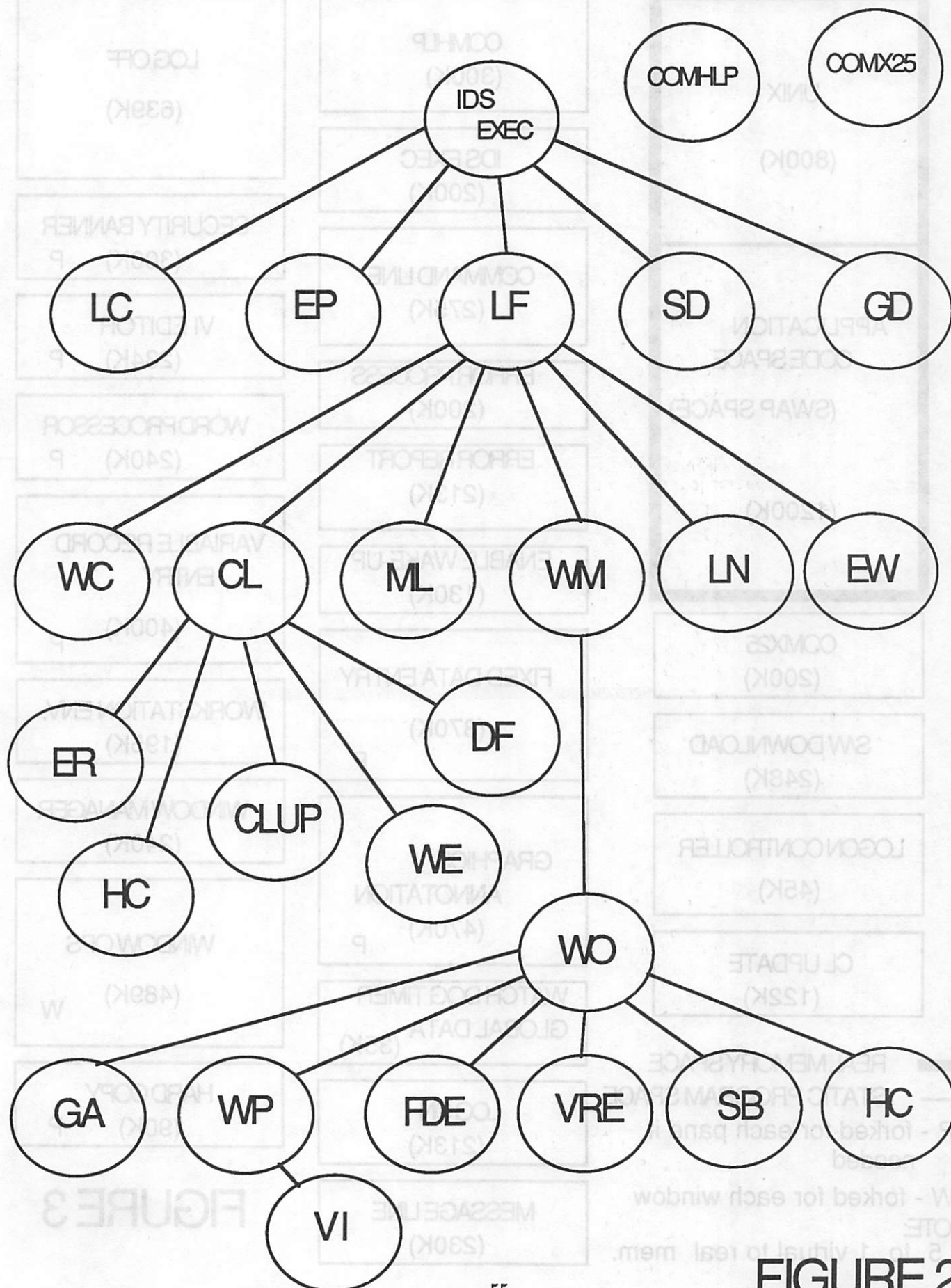
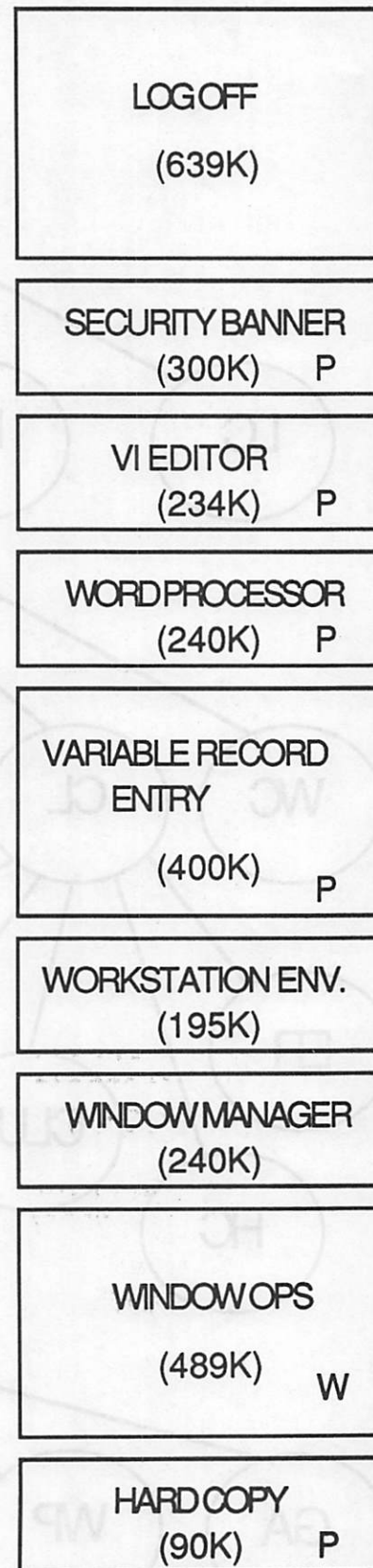
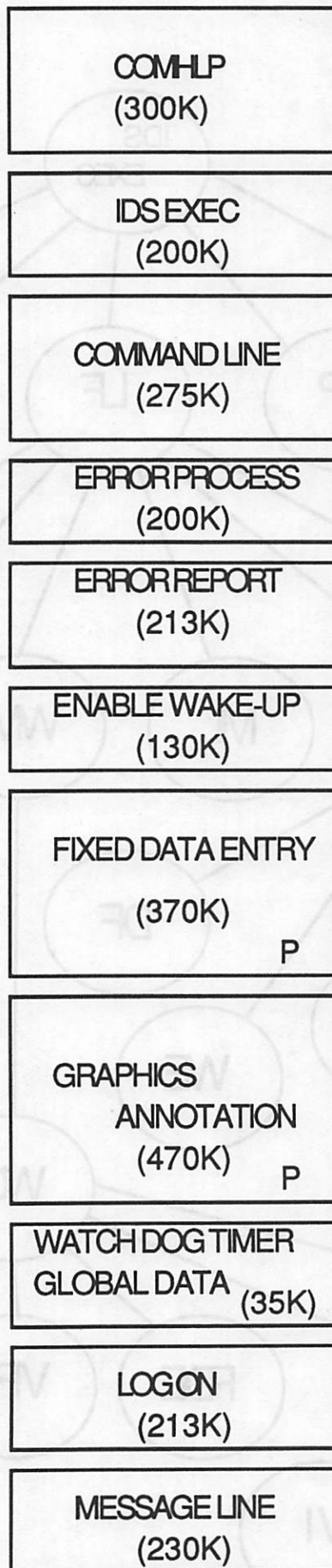
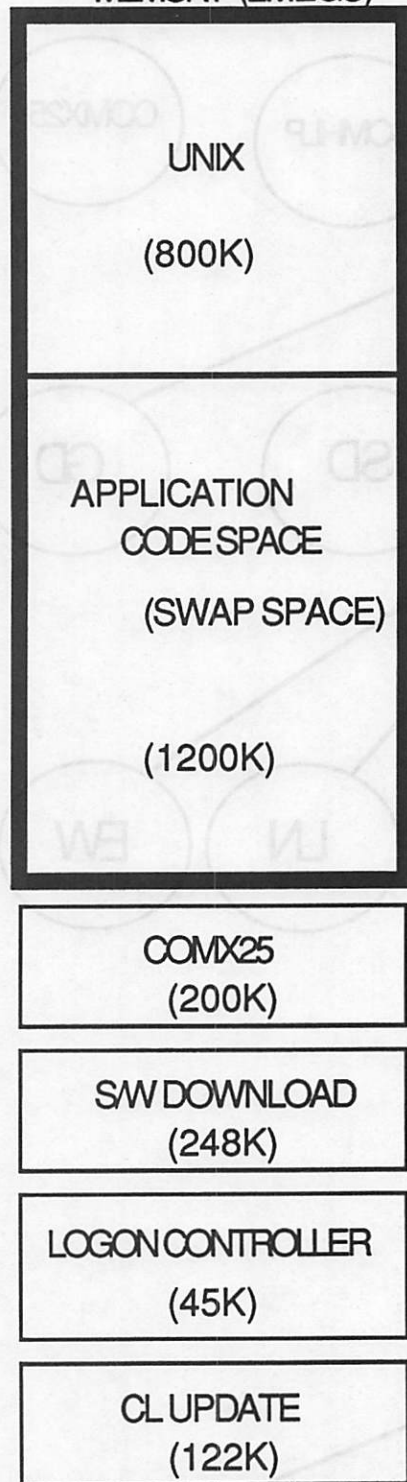


FIGURE 2

# MEMORY (2MEGS)



**REAL MEMORY SPACE**  
**STATIC PROGRAM SPACE**  
 P - forked for each pane if needed  
 W - forked for each window  
 NOTE:  
 5 to 1 virtual to real mem.

## FIGURE 3

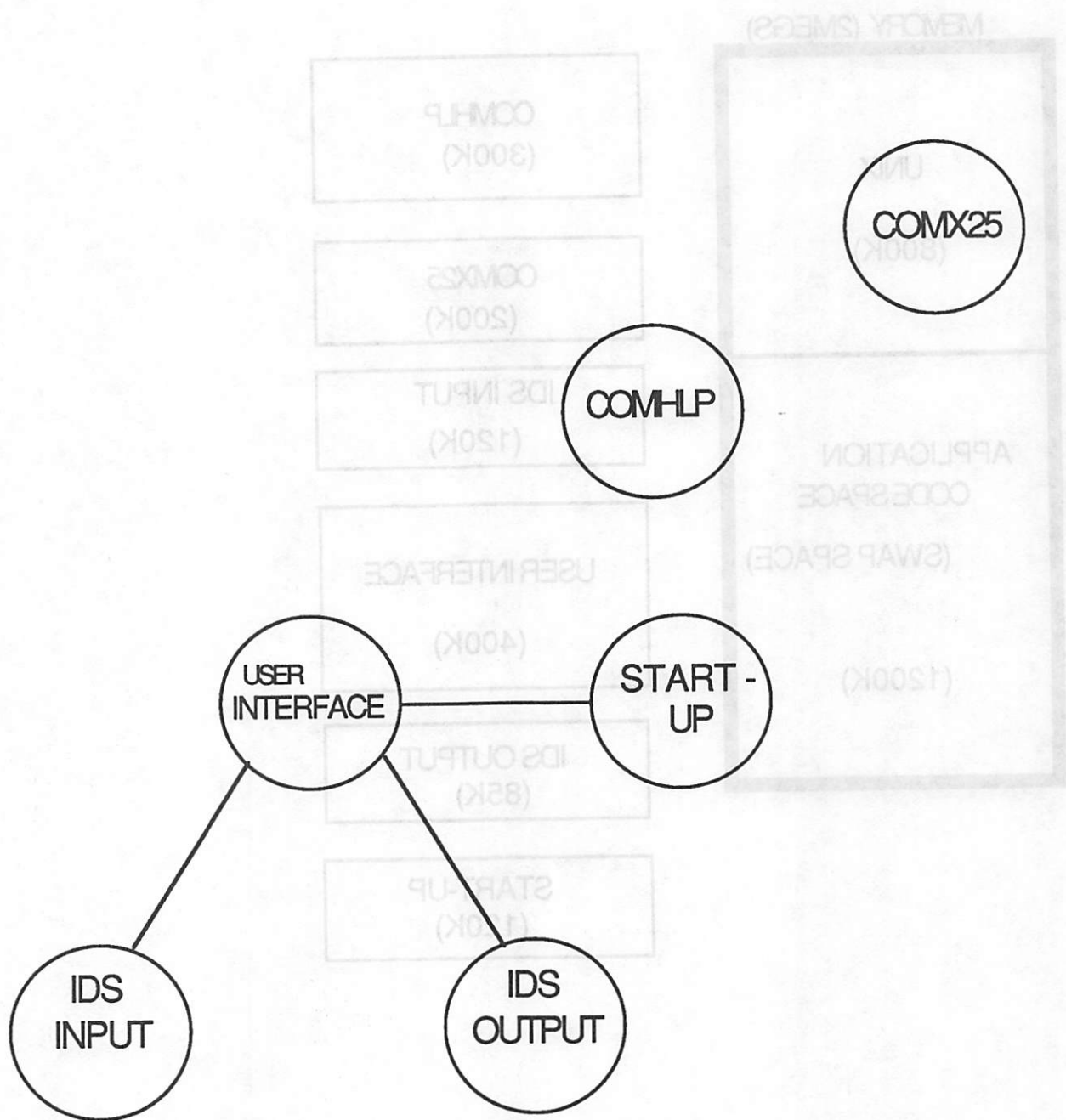
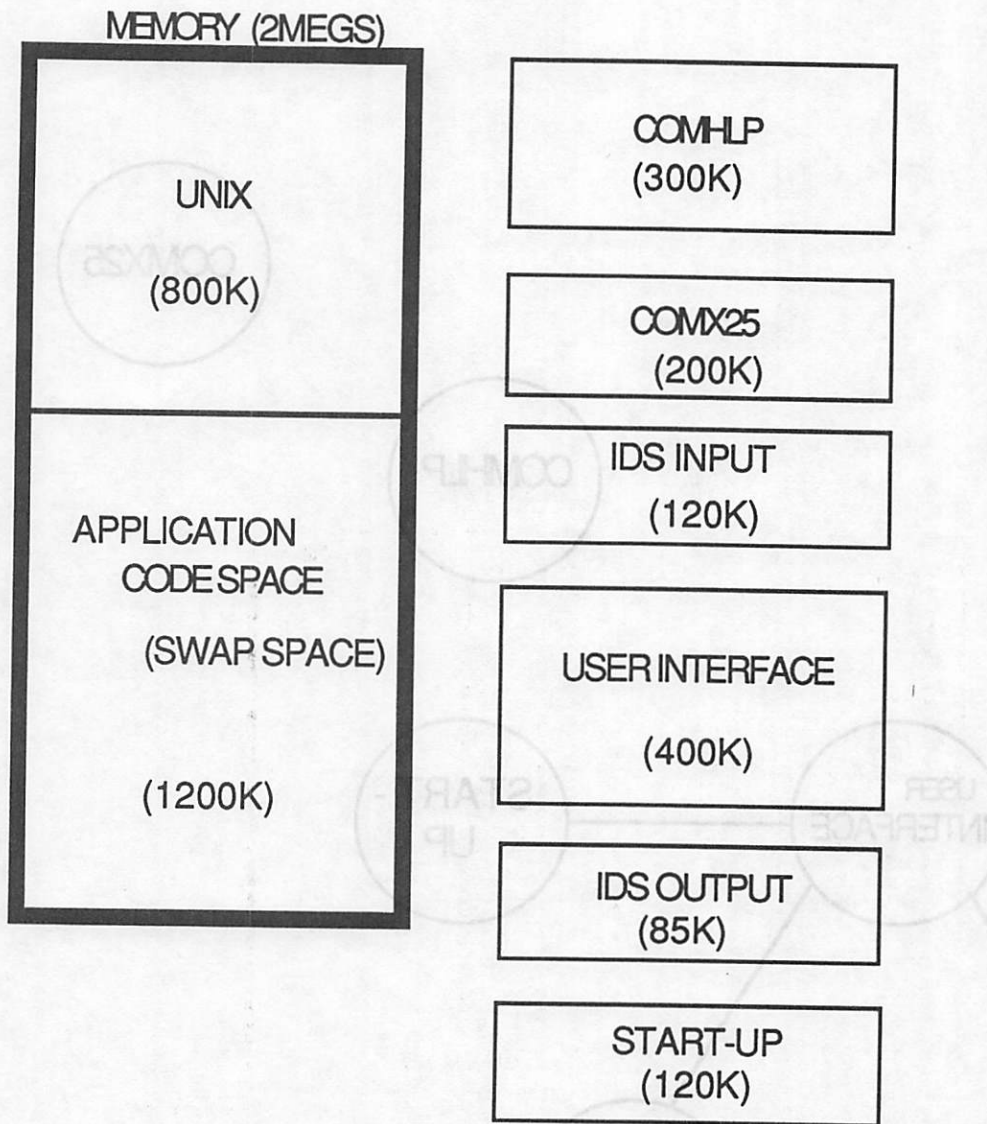


FIGURE 4

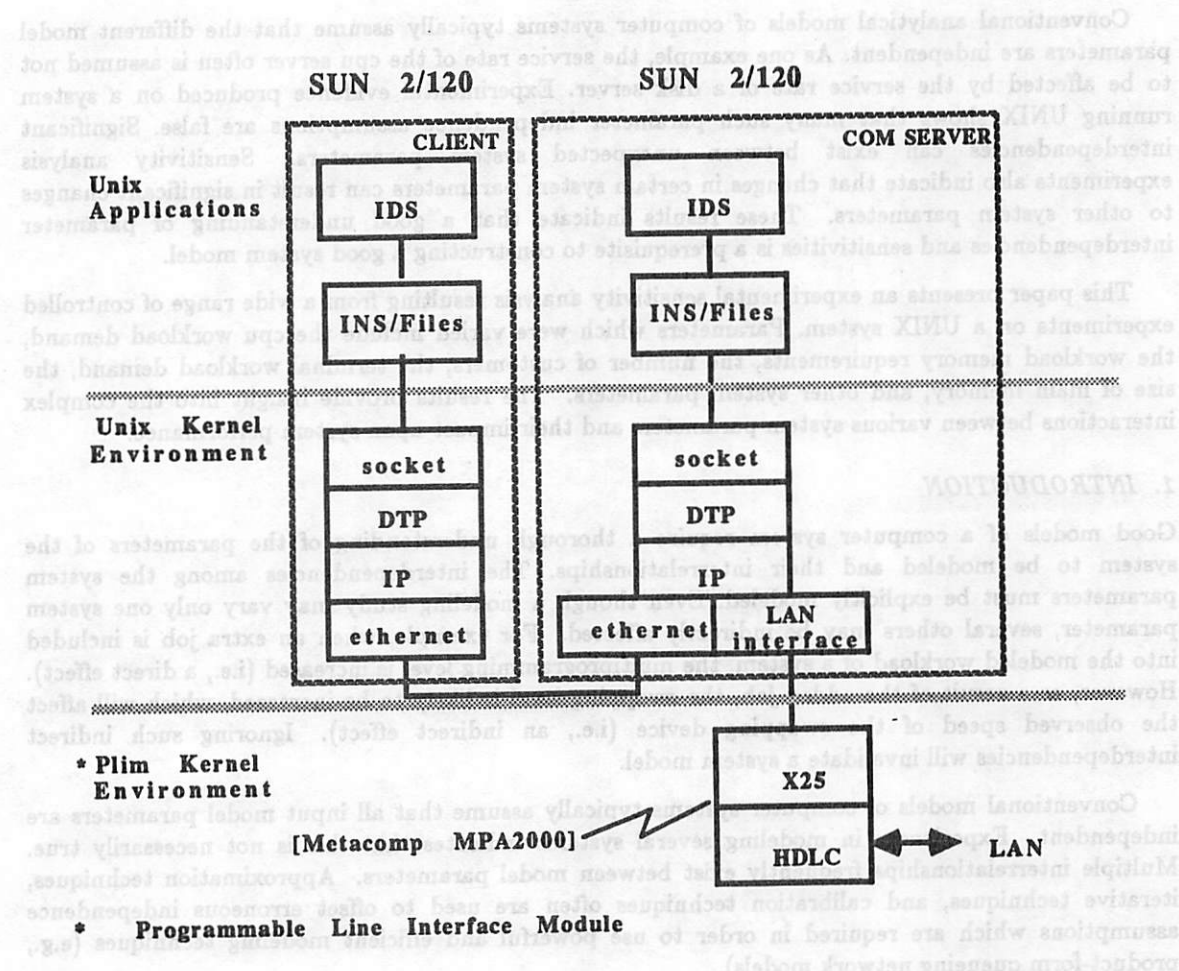


— REAL MEMORY SPACE  
— STATIC PROGRAM SPACE

FIGURE 5

Lindsay E. Stephens and Lawrence W. Dowdy  
Computer Science Department  
Vanderbilt University  
Box 1875, Station B  
Nashville, TN 37235

ABSTRACT



**FIGURE 6**  
**"X25 REHOST" COMMUNICATION ARCHITECTURE**

# Experimental Sensitivity Analysis of Performance in a UNIX System

Lindsey E. Stephens and Lawrence W. Dowdy

Computer Science Department  
Vanderbilt University  
Box 1679, Station B  
Nashville, TN 37235

## ABSTRACT

Conventional analytical models of computer systems typically assume that the different model parameters are independent. As one example, the service rate of the cpu server often is assumed not to be affected by the service rate of a disk server. Experimental evidence produced on a system running UNIX shows that many such parameter independence assumptions are false. Significant interdependencies can exist between unexpected system parameters. Sensitivity analysis experiments also indicate that changes in certain system parameters can result in significant changes to other system parameters. These results indicate that a good understanding of parameter interdependencies and sensitivities is a prerequisite to constructing a good system model.

This paper presents an experimental sensitivity analysis resulting from a wide range of controlled experiments on a UNIX system. Parameters which were varied include the cpu workload demand, the workload memory requirements, the number of customers, the terminal workload demand, the size of main memory, and other system parameters. The results provide insight into the complex interactions between various system parameters and their impact upon system performance.

## 1. INTRODUCTION

Good models of a computer system require a thorough understanding of the parameters of the system to be modeled and their interrelationships. The interdependencies among the system parameters must be explicitly modeled. Even though a modeling study may vary only one system parameter, several others may be indirectly affected. For example, when an extra job is included into the modeled workload of a system, the multiprogramming level is increased (i.e., a direct effect). However, as a result of the added job, the swapping level is likely to be increased, which will affect the observed speed of the swapping device (i.e., an indirect effect). Ignoring such indirect interdependencies will invalidate a system model.

Conventional models of computer systems typically assume that all input model parameters are independent. Experiences in modeling several systems indicates that this is not necessarily true. Multiple interrelationships frequently exist between model parameters. Approximation techniques, iterative techniques, and calibration techniques often are used to offset erroneous independence assumptions which are required in order to use powerful and efficient modeling techniques (e.g., product-form queueing network models).

A different approach to building better models is to concentrate on identifying all significant interdependencies among system parameters. Then a detailed analysis of the sensitivity of each of these relationships can be used to build a composite model of the system.

A detailed experimental analysis of the interrelationships among various system parameters on a system running UNIX is described below. This analysis includes both an identification and sensitivity analysis. The purpose of the experiments is to provide direction and data for future theoretical analysis, validation of previous theory, and guidance for the construction of better practical models. These experiments also provide proof that significant interdependencies exist among system parameters. Thus, parameter interdependencies need to be identified, better understood, and modeled.

Section 2 briefly describes the hardware and software environment used for the experiments. Section 3 presents the parameters studied, the experiments conducted, and the specific observations made. Section 4 summarizes the results.

## 2. ENVIRONMENT

The experimental system hardware is a Perkin-Elmer 3220 running UNIX edition 2.7. The system has 960K bytes of main memory and has two 70M byte disks on separate controllers. Each disk is partitioned into several logical disks. Each logical subdisk appears to be an independent disk at the user level. One disk contains two logical disks: 1) *arcv*, which contains manuals and online short-term backup of users' files; and 2) *usr*, which contains system libraries, mail files, and other system files useful to users. (This disk has a small, constant demand placed upon it so it is ignored in the experimental analysis.) The other disk contains three logical disks: 1) *root*, which contains system files, system and command binary object files, and temporary storage space; 2) *users*, which contains user files; and 3) *swap*, which is the system swap area.

The experimental system software consists of a synthetic benchmarking facility and a system performance monitor. The synthetic benchmarking facility consists of a benchmark generator and a benchmark executor. The benchmark generator generates a user-specified, distribution driven, synthetic benchmark. A typical synthetic benchmark consists of several independent jobs. Each job has a specified size, a specified file access pattern, a specified cpu demand distribution, and a specified idle time distribution (i.e., to mimic the think times of terminal users). All variable parameters, such as think times, cpu burn times, and disk branching probabilities, may be varied according to the input. In these experiments all parameters not being varied were parameterized from data gathered on the system over a several month period during normal user hours. Thus, the generated synthetic benchmarks are representative of actual observed workloads.

The benchmark executor is used to control the distribution of the number of synthetic benchmarks (i.e., the distribution of the multiprogramming level, MPL) active during the monitoring period. The executor allows the use of distinctly parameterized synthetic benchmarks which provides flexibility in workload generation. The benchmark executor also synchronizes with the performance monitor.

The performance monitor consists of a system part and a user part. The system part monitors system parameters (e.g., cpu utilization, disk busy times, block counts, etc.) continuously when monitoring is enabled. The system interface consists of a single system call with facilities for beginning or ending monitoring, choosing which certain parameters are to be monitored, and copying monitored data to the user address space.

The user part of the performance monitor performs monitoring setup and final data transfer using the system interface. Upon conclusion of monitoring, the user part also performs data reduction and formats the requested output reports. The output reports include device utilizations, throughputs, speeds, and queue lengths. Disk busy times are broken down into seek and transfer times. Cpu utilizations are broken down into system and user times. A loop counter is incorporated into the benchmark generator to give a measure of useful throughput for comparison against raw cpu throughput which includes overhead activities. Several other parameters are monitored but are not reported here.

The synthetic benchmark is written to allow incorporation of system and benchmark changes in an systematic manner. The overhead of the performance monitor is less than three per cent in all configurations. The overhead of the benchmark executor is also low in most configurations but can incur a high cost in signalling, forking, and synchronization for high MPLs.

## 3. EXPERIMENTAL SENSITIVITY

The following methodology is used. Each parameter, or its distribution, is varied while all other parameters are held constant according to the base representative workload. A range of values is

selected for each parameter. This range is designed to roughly include the interval from a zero value for the parameter to twice its normal value as determined from the base representative workload. In several instances, requirements of integral parameter values and matching coefficients of distributions result in somewhat different ranges with uneven steps. Each experiment consists of running one synthetic workload, consisting of several jobs, along with the performance monitor for thirty minutes. A five minute unmonitored startup time minimizes transient startup effects.

The following parameters are varied in the experiments:

- the large process cutoff,
- the mean cpu workload demand,
- the distribution of the cpu workload demand,
- the mean MPL,
- the distribution of the MPL,
- the mean size of main memory,
- the distribution of main memory size,
- the mean terminal workload demand, and
- the distribution of the terminal workload demand.

In the following, all tables list the experiments in order of increasing base parameter value. The distributional quantity of main interest is the coefficient of variation (CV), the standard deviation divided by the mean. The CV is a measure of the spread of a distribution about its mean. Higher moments were also affected by changing the distributions but the text implicitly refers to the CV when describing a distribution unless explicitly stated.

### *Large Process Cutoff*

The large process cutoff is an internal parameter used by the scheduler to determine which processes to swap. Processes smaller than the cutoff gain preferential swapping treatment over processes larger than the cutoff. The effect is to partition the workload into two classes, "small" and "large" jobs. As the cutoff is increased, the number of small jobs increases while the number of large jobs decreases. This parameter is effective in preventing thrashing. Seven experiments were run with a workload of six processes. The number of large processes is varied from zero to six. The results are summarized in Table 1.

**TABLE 1. Sensitivity to large process cutoff**

%Change	Cpu						Disk		
	UsUti	SyUti	Util	UsTput	Speed	Tput	Util	Speed	Tput
-100.00	26.5	3.8	30.3	0.6835	16.0	4.8	16.8	26.3	4.4
-66.67	29.6	4.4	33.9	0.7610	16.0	5.4	19.0	26.2	5.0
-33.33	31.5	4.7	36.2	0.8111	15.9	5.8	20.6	25.7	5.3
0.00	34.3	5.1	39.4	0.8905	16.0	6.3	22.4	25.9	5.8
33.33	34.7	5.4	40.1	0.9071	18.8	7.5	31.8	22.0	7.0
66.67	31.5	5.7	37.2	0.8748	32.5	12.1	72.0	16.1	11.6
100.00	29.0	6.3	35.3	0.8608	40.9	14.4	89.1	15.6	13.9

%Change	Root				Users			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	3639	17.07	11.96	29.03	3236	21.87	11.88	33.75
-66.67	4006	17.62	11.43	29.06	3613	22.49	11.91	34.40
-33.33	4246	18.17	11.77	29.93	3851	22.78	11.69	34.46
0.00	4643	17.87	11.66	29.53	4189	22.98	11.80	34.79
33.33	4759	21.02	11.60	32.62	4279	24.29	11.52	35.82
66.67	4597	33.07	10.97	44.04	4154	29.38	10.29	39.67
100.00	4537	39.42	10.19	49.61	4047	31.57	10.04	41.61

%Change	Swap				Total			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	1150	15.69	62.43	78.12	8025	18.81	19.16	37.97
-66.67	1347	15.66	59.48	75.15	8966	19.29	18.85	38.13
-33.33	1417	15.53	62.16	77.68	9514	19.64	19.24	38.88
0.00	1631	14.73	59.30	74.03	10463	19.43	19.14	38.57
33.33	3526	17.93	56.83	74.76	12564	21.27	24.27	45.54
66.67	12081	18.24	58.57	76.81	20832	23.74	38.44	62.17
100.00	16504	17.56	55.80	73.37	25088	23.77	40.17	63.95

Abbreviations		
UsUti	-	User cpu utilization
SyUti	-	System cpu utilization
Util	-	Total device utilization
UsTput	-	Useful throughput (average per second)
Speed	-	Device service rate (average per second)
Tput	-	Device throughput
Req	-	Number of disk I/O requests
Sk/Rq	-	Disk seek time per request
Tr/Rq	-	Disk transfer time per request
Bu/Rq	-	Disk busy (seek plus transfer) time per request

The following observations are made.

1. Given an X% change in the magnitude of the large process cutoff parameter, other system parameters and performance parameters can change by more the X%. This is true for the cpu speed, overall disk utilization, and mean seek time on root.
2. Useful throughput and cpu utilization are concave down with respect to the number of small processes. The four small processes fit into main memory while the two large processes stay swapped out during much of the monitoring period.
3. Swapping disk utilization increases slowly and monotonically with respect to the number of small processes until the size of those processes is 60% of available main memory. Further increases in the cutoff cause the swapping disk to become a bottleneck.

4. The cpu speed is constant with four or fewer small processes. Five or more small processes cause the cpu speed to increase superlinearly. This is due to a close dependence of the cpu speed on the amount of swapping and the sizes of the swapped processes.
5. These experiments indicate that the performance of the system is quite sensitive to the large process cutoff parameter value. They indicate that the cutoff should be set to a value such that the total memory demand of the small processes is approximately 60% of available main memory.

#### CPU Workload Demand: Mean and Distribution

The cpu workload demand specifies how long a cpu burn occurs between think times and/or disk input/output. A loop in the benchmark generator is parameterized by a continuous Coxian server distribution with mean varying from 0.0 to 2.0 (-100% to +100% change) and with a coefficient of variation, CV (standard deviation /mean), varying from 0.0 to 10.0 (-100% to +233% change). The results are summarized in Tables 2 and 3.

Total				Swap				Cutoff
Req	Tr/Rq	SE/Rq	CV	Req	Tr/Rq	SE/Rq	CV	
100.00	10.19	49.43	4.87	100.00	10.19	49.43	4.87	100.00
66.67	10.97	38.07	3.47	66.67	10.97	38.07	3.47	66.67
33.33	11.00	27.03	2.43	33.33	11.00	27.03	2.43	33.33
0.00	11.87	17.87	1.51	0.00	11.87	17.87	1.51	0.00
-33.33	12.17	12.17	1.00	-33.33	12.17	12.17	1.00	-33.33
-66.67	12.43	9.98	0.80	-66.67	12.43	9.98	0.80	-66.67
-100.00	12.60	8.02	0.63	-100.00	12.60	8.02	0.63	-100.00

Total				Swap				Cutoff
Req	Tr/Rq	SE/Rq	CV	Req	Tr/Rq	SE/Rq	CV	
100.00	10.19	49.43	4.87	100.00	10.19	49.43	4.87	100.00
66.67	10.97	38.07	3.47	66.67	10.97	38.07	3.47	66.67
33.33	11.00	27.03	2.43	33.33	11.00	27.03	2.43	33.33
0.00	11.87	17.87	1.51	0.00	11.87	17.87	1.51	0.00
-33.33	12.17	12.17	1.00	-33.33	12.17	12.17	1.00	-33.33
-66.67	12.43	9.98	0.80	-66.67	12.43	9.98	0.80	-66.67
-100.00	12.60	8.02	0.63	-100.00	12.60	8.02	0.63	-100.00

Req	Tr/Rq	SE/Rq	CV	Req	Tr/Rq	SE/Rq	CV	Cutoff
100.00	10.19	49.43	4.87	100.00	10.19	49.43	4.87	100.00
66.67	10.97	38.07	3.47	66.67	10.97	38.07	3.47	66.67
33.33	11.00	27.03	2.43	33.33	11.00	27.03	2.43	33.33
0.00	11.87	17.87	1.51	0.00	11.87	17.87	1.51	0.00
-33.33	12.17	12.17	1.00	-33.33	12.17	12.17	1.00	-33.33
-66.67	12.43	9.98	0.80	-66.67	12.43	9.98	0.80	-66.67
-100.00	12.60	8.02	0.63	-100.00	12.60	8.02	0.63	-100.00

**TABLE 2. Sensitivity to the mean cpu workload demand**

%Change	Cpu						Disk		
	UsUti	SyUti	Util	UsTput	Speed	Tput	Util	Speed	Tput
-100.00	1.8	9.9	11.7	1.3861	135.1	15.8	48.9	29.3	14.4
-80.00	24.0	9.9	33.9	1.3319	42.8	14.5	43.2	30.4	13.1
-60.00	41.6	9.8	51.4	1.1689	27.1	13.9	45.1	28.2	12.7
-40.00	55.0	9.3	64.3	1.0487	19.6	12.6	45.4	25.3	11.5
-20.00	70.3	7.9	78.2	1.0073	13.3	10.4	26.6	35.2	9.3
0.00	77.7	7.5	85.2	0.8912	11.4	9.7	27.8	31.7	8.8
20.00	84.9	6.9	91.8	0.8100	10.0	9.1	27.0	30.7	8.3
40.00	90.3	5.7	95.9	0.7489	8.0	7.7	19.8	35.0	6.9
60.00	90.5	6.0	96.5	0.6515	8.6	8.3	27.9	27.0	7.5
80.00	94.7	4.6	99.3	0.6043	6.4	6.3	16.6	34.4	5.7
100.00	93.8	4.4	98.2	0.5425	5.7	5.6	14.4	34.8	5.0

%Change	Root				Users			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	11832	16.41	11.01	27.42	10819	19.75	10.91	30.66
-80.00	11265	15.80	11.07	26.87	10274	20.02	10.88	30.90
-60.00	10070	17.46	10.95	28.41	9161	20.92	10.87	31.79
-40.00	8894	19.23	10.65	29.88	8179	22.11	10.58	32.69
-20.00	8661	14.54	10.81	25.34	7915	20.16	10.57	30.73
0.00	7696	15.71	10.73	26.45	6941	20.82	10.40	31.21
20.00	7050	15.99	10.57	26.56	6262	21.32	10.30	31.62
40.00	6539	14.45	10.63	25.07	5702	20.40	10.39	30.80
60.00	5668	18.17	10.61	28.78	5020	22.59	10.21	32.80
80.00	5319	15.03	10.73	25.76	4745	21.20	10.41	31.61
100.00	4721	14.76	11.06	25.82	4295	21.90	10.01	31.90

%Change	Swap				Total			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	3160	25.87	44.96	70.82	25811	18.97	15.12	34.09
-80.00	2065	29.00	47.30	76.30	23604	18.79	14.15	32.95
-60.00	3670	24.29	39.67	63.96	22901	19.94	15.52	35.46
-40.00	3624	29.51	48.79	78.29	20697	22.17	17.30	39.47
-20.00	240	23.75	40.00	63.75	16816	17.31	11.11	28.43
0.00	1179	24.61	42.63	67.24	15816	18.62	12.96	31.58
20.00	1614	21.49	40.92	62.40	14926	18.82	13.74	32.56
40.00	252	24.52	44.52	69.05	12493	17.37	11.20	28.57
60.00	2884	19.85	40.64	60.49	13572	20.16	16.84	37.01
80.00	205	16.98	40.49	57.46	10269	17.92	11.18	29.10
100.00	0	0.00	0.00	0.00	9016	18.16	10.56	28.72

**TABLE 3. Sensitivity to the cpu workload demand distribution (CV)**

%Change	Cpu						Disk		
	UsUti	SyUti	Util	UsTput	Speed	Tput	Util	Speed	Tput
-100.00	51.6	8.9	60.5	1.2140	21.2	12.8	30.3	38.0	11.5
-20.00	53.8	8.9	62.7	1.2509	20.7	13.0	29.8	39.1	11.7
0.00	52.1	9.2	61.3	1.1977	21.6	13.3	39.4	30.4	12.0
66.67	52.6	10.2	62.8	1.1663	25.5	16.0	62.7	23.6	14.8
233.33	51.4	10.3	61.7	1.2126	27.1	16.7	64.8	23.9	15.4

%Change	Root				Users			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	10673	14.22	8.60	22.82	9389	19.60	8.38	27.98
-20.00	10886	14.18	7.89	22.08	9667	19.41	7.79	27.20
0.00	10438	15.68	10.90	26.58	9181	20.18	10.82	30.99
66.67	10034	21.12	10.70	31.82	8825	22.27	10.67	32.95
233.33	21051	20.55	10.76	31.31	18512	21.65	10.86	32.51

%Change	Swap				Total			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	641	22.03	38.16	60.19	20703	16.90	9.42	26.32
-20.00	465	27.48	45.16	72.65	21018	16.88	8.67	25.55
0.00	1911	30.30	46.75	77.05	21530	18.89	14.05	32.94
66.67	7738	25.86	41.12	66.98	26597	22.88	19.54	42.42
233.33	16064	24.15	42.54	66.70	55627	21.96	19.97	41.93

The following observations are made.

1. An X% change in the mean cpu workload demand corresponds to roughly an X% change in the cpu throughput and in the useful throughput. The agrees with queueing network model theoretical results.
2. Cpu utilization increases steadily as a function of the mean cpu demand. Since the cpu demand increases, disk demand (e.g., utilization and throughput) correspondingly decrease. These results are expected.
3. As the mean cpu demand increases, the mean seek time per request and per block on the swap subdisk generally decreases due to decreasing requests to root and users. This is in turn caused by longer cpu burn times which decrease the amount of process switching.
4. As the CV of the cpu demand increases, total cpu throughput and disk throughput increase. This is contrary to theoretical studies which indicate that best performance occurs when the CV is low.
5. In general, as the CV of the cpu demand increases, the amount of context switching and swapping increases. This has the effect of slowing down the disk and therefore increasing its utilization and throughput.
6. The most significant effects are with high CVs (e.g., CV=10, the last experiment). A high CV increases disk activity substantially. Thus, in highly variable workloads, knowing only the mean cpu demand and ignoring its CV may lead to erroneous models.

#### *Multiprogramming Level: Mean and Distribution*

The multiprogramming level specifies the number of active synthetic jobs concurrently running on the system. The multiprogramming level changes from a mean of 0 to 40 (-100% to +60% change) and the CV of multiprogramming level distribution changes from 0.2 to 1.8 (-67% to +200% change). The results are summarized in Tables 4 and 5. In Table 4 only, the useful throughput reported is the gross aggregate per second rather than the average per process.

TABLE 4. Sensitivity to the mean multiprogramming level

%Change	Cpu						Disk		
	UsUti	SyUti	Util	UsTput	Speed	Tput	Util	Speed	Tput
-100.00	0.0	0.3	0.3	0.0000	9.2	0.0	0.0	59.1	0.0
-80.00	27.3	4.2	31.5	7.0970	23.2	7.3	16.9	38.9	6.6
-60.00	49.1	9.2	58.3	10.8880	22.4	13.0	30.2	38.6	11.7
-40.00	60.2	14.6	74.7	15.7680	21.7	16.2	36.9	38.3	14.1
-20.00	43.2	21.3	64.5	10.9220	26.5	17.1	75.7	21.1	15.9
0.00	36.6	23.8	60.4	8.8650	29.4	17.8	91.9	18.3	16.8
20.00	39.7	26.7	66.3	9.1410	27.0	17.9	93.5	18.1	16.9
40.00	45.2	29.4	74.6	9.7930	24.0	17.9	90.5	18.6	16.9
60.00	45.6	30.0	75.6	11.1760	23.8	18.0	90.3	18.7	16.9

%Change	Root				Users			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	13	4.61	12.31	16.92	0	0.00	0.00	0.00
-80.00	6386	13.46	9.71	23.17	5434	18.94	9.73	28.67
-60.00	11112	13.67	9.66	23.34	9900	19.11	9.62	28.73
-40.00	13446	13.88	9.66	23.53	11934	19.41	9.57	28.98
-20.00	9391	27.87	9.77	37.64	8476	26.40	9.69	36.09
0.00	7576	39.00	9.99	48.99	6749	31.85	9.91	41.76
20.00	7803	39.90	9.81	49.70	6879	32.97	9.90	42.87
40.00	8503	37.19	9.98	47.17	7380	32.31	9.78	42.10
60.00	8576	37.01	9.95	46.96	7471	32.57	9.85	42.41

%Change	Swap				Total			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	0	0.00	0.00	0.00	13	4.61	12.31	16.92
-80.00	6	20.00	26.67	46.67	11826	15.98	9.73	25.71
-60.00	0	0.00	0.00	0.00	21012	16.24	9.64	25.88
-40.00	42	20.00	13.33	33.33	25422	16.48	9.62	26.10
-20.00	10821	29.94	35.03	64.98	28688	28.22	19.28	47.49
0.00	15936	29.95	32.91	62.86	30261	32.64	22.04	54.68
20.00	15812	31.07	32.20	63.27	30494	33.76	21.44	55.20
40.00	14517	31.90	31.38	63.28	30400	33.48	20.15	53.63
60.00	14375	32.08	30.96	63.04	30422	33.59	19.85	53.44

**TABLE 5. Sensitivity to the multiprogramming level distribution (CV)**

%Change	Cpu						Disk		
	UsUti	SyUti	Util	UsTput	Speed	Tput	Util	Speed	Tput
-66.67	28.3	4.4	32.7	1.3553	21.4	7.0	14.8	41.0	6.1
-53.33	28.2	4.5	32.6	1.3932	21.1	6.9	14.4	41.3	5.9
0.00	25.2	4.8	30.1	1.1372	23.1	6.9	20.0	30.6	6.1
66.67	19.6	7.0	26.5	1.0648	26.0	6.9	23.6	22.9	5.4
100.00	27.2	8.8	36.0	0.7963	30.8	11.1	44.5	20.1	8.9
200.00	32.9	11.7	44.6	0.4872	30.9	13.8	61.7	18.4	11.4

%Change	Root				Users			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-66.67	5812	13.68	7.95	21.63	4929	19.15	7.83	26.98
-53.33	5613	13.71	7.63	21.34	4987	19.06	7.75	26.81
0.00	5057	17.07	8.03	25.10	4368	20.92	7.82	28.74
66.67	3769	22.52	8.35	30.87	3149	23.36	8.02	31.38
100.00	5232	28.73	9.10	37.82	4258	26.85	8.82	35.67
200.00	5748	34.32	9.59	43.90	4835	30.48	9.41	39.88

%Change	Swap				Total			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-66.67	147	12.65	35.65	48.30	10888	16.14	8.27	24.41
-53.33	101	17.03	38.22	55.25	10701	16.23	7.98	24.21
0.00	1580	25.11	42.84	67.95	11005	19.75	12.94	32.70
66.67	2819	28.41	45.92	74.33	9737	24.50	19.12	43.62
100.00	6603	26.08	42.31	68.38	16093	27.14	22.65	49.79
200.00	9873	26.61	40.76	67.38	20456	29.69	24.59	54.28

The following observations are made.

1. As the mean MPL increases, cpu utilizations and disk throughput increase monotonically. At low MPL values a change of X% in the mean MPL causes these performance parameters to increase by roughly X%. At high MPL values the rate of increase is lower.
2. As the mean MPL increases useful throughput increases until the swapping disk becomes a bottleneck. Thrashing then begins and the useful throughput drops by roughly 30% where it levels off.
3. As the mean MPL increases, the speed of the disk drops and the mean seek time per request increases significantly. These parameter changes are usually unmodelled but are explained by the increase in swapping activity.
4. As the CV of the MPL distribution increases, useful throughput decreases, the cpu speed increases, the disk utilization increases, the disk speed increases, and the amount of swapping increases. These changes are significant but are rarely modelled.
5. The useful throughput is correlated the the moments of the distribution. Generally, as the first moment increases, performance increases. As the second moment increases, performance decreases. As the third moment increases, performance increases. These correlations are fairly significant.

#### *Main Memory Size: Mean and Distribution*

The main memory size is the amount of main memory available for user processes. The mean memory size is changed from 240K to 960K (-57% to +71% change) and its CV is changed from 0.0 to 0.77 (-100% to +26%). The results are summarized in Tables 6 and 7.

**TABLE 6. Sensitivity to the mean main memory size**

Cpu							Disk		
%Change	UsUti	SyUti	Util	UsTput	Speed	Tput	Util	Speed	Tput
-57.14	35.6	10.3	45.9	0.6454	32.9	15.1	57.2	25.3	14.4
-42.86	37.2	9.0	46.2	0.7184	37.3	17.2	81.8	20.1	16.5
-14.28	35.7	9.0	44.6	0.7467	39.3	17.5	93.6	17.9	16.7
0.00	36.3	9.0	45.3	0.7639	37.6	17.1	93.3	17.4	16.3
14.29	35.6	9.2	44.8	0.7640	37.4	16.8	92.1	17.3	15.9
42.86	37.1	9.7	46.8	0.8150	35.6	16.7	85.2	18.6	15.8
71.43	50.6	9.3	59.9	1.1353	22.7	13.6	42.3	29.4	12.4

Root					Users			
%Change	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-57.14	5592	19.45	8.86	28.31	5003	20.59	8.69	29.28
-42.86	6199	33.52	9.67	43.19	5558	30.67	9.26	39.92
-14.28	6439	41.18	9.91	51.09	5762	33.35	9.52	42.86
0.00	6573	40.56	9.72	50.28	5906	32.93	9.45	42.38
14.29	6562	40.18	9.56	49.73	5874	32.19	9.41	41.59
42.86	7057	34.79	9.00	43.79	6339	29.03	8.86	37.89
71.43	9800	17.92	8.04	25.96	8953	21.27	7.97	29.24

Swap					Total			
%Change	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-57.14	15378	10.20	36.87	47.07	25973	14.20	25.41	39.61
-42.86	17891	16.82	38.13	54.95	29648	22.91	26.76	49.67
-14.28	17928	23.16	38.68	61.84	30129	28.96	26.96	55.92
0.00	16791	24.99	40.37	65.36	29270	30.09	27.25	57.34
14.29	16233	25.32	41.61	66.93	28669	30.13	27.67	57.80
42.86	15087	23.48	41.83	65.30	28483	27.52	26.36	53.87
71.43	3647	25.45	41.79	67.24	22400	20.49	13.51	33.99

**TABLE 7. Sensitivity to the main memory size distribution (CV)**

Cpu							Disk		
%Change	UsUti	SyUti	Util	UsTput	Speed	Tput	Util	Speed	Tput
-100.00	76.4	7.8	84.3	0.9193	11.6	9.8	27.6	32.1	8.9
0.00	63.8	9.0	72.8	0.7647	17.2	12.5	61.3	19.1	11.7
26.49	69.6	8.5	78.1	0.8148	15.2	11.9	49.9	22.2	11.1

Root					Users			
%Change	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	8095	15.04	12.00	27.04	7136	20.44	11.82	32.26
0.00	6544	27.00	11.38	38.38	5814	25.63	11.20	36.84
26.49	7226	22.01	11.82	33.82	6374	23.88	11.56	35.44

Swap					Total			
%Change	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	726	24.16	40.88	65.04	15957	17.87	13.23	31.10
0.00	8744	24.12	48.90	73.02	21102	25.43	26.88	52.31
26.49	6300	22.32	45.53	67.85	19900	22.71	22.41	45.12

The following observations are made.

1. As the mean memory size increases, useful throughput increases monotonically. This increase can be significant. In one case, a 25% increase in main memory leads to a 40% increase in useful throughput.

2. However, as the mean memory size increases, disk seek time increases initially and then decreases. The same behavior is exhibited by cpu speed, total cpu throughput, disk utilization, and disk throughput. These interdependencies are usually not modelled.
3. As the mean memory size increases, the disk speeds generally decrease initially and then increase.
4. The CV of the memory size distribution does affect the other parameters. However, more tests are necessary to detect any consistent trends.

#### Terminal Workload Demand: Mean and Distribution

The terminal workload demand is the amount of time that each synthetic job spends sleeping. This mimics the think time distribution imposed by interactive users. The mean terminal workload demand is varied from 0 to 20 seconds (-100% to +300%). The distribution of the terminal workload demand is varied from a CV of 0 to 3 (-100% to +200%). The results are summarized in Tables 8 and 9.

**TABLE 8. Sensitivity to the mean terminal workload demand**

%Change	Cpu						Disk		
	UsUti	SyUti	Util	UsTput	Speed	Tput	Util	Speed	Tput
-100.00	68.4	14.9	83.3	1.5596	19.2	16.0	33.8	42.4	14.4
0.00	51.0	9.3	60.3	1.1435	21.8	13.1	38.4	31.1	12.0
100.00	29.8	6.3	36.1	0.6778	27.4	9.9	41.0	22.4	9.2
300.00	16.4	3.7	20.1	0.3696	29.0	5.9	24.4	22.4	5.4

%Change	Root				Users			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	13175	13.41	6.94	20.34	12027	17.94	6.67	24.61
0.00	9899	17.00	7.64	24.65	9003	20.92	7.88	28.80
100.00	5842	23.31	8.24	31.55	5194	23.00	8.10	31.10
300.00	3254	23.04	8.53	31.57	2933	22.39	7.92	30.31

%Change	Swap				Total			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	622	35.30	36.30	71.61	25824	16.04	7.52	23.57
0.00	2614	26.83	44.89	71.72	21516	19.84	12.27	32.10
100.00	5478	26.68	44.90	71.58	16514	24.33	20.36	44.69
300.00	3621	22.37	45.75	68.12	9808	22.60	22.09	44.69

**TABLE 9. Sensitivity to the terminal workload demand distribution (CV)**

%Change	Cpu						Disk		
	UsUti	SyUti	Util	UsTput	Speed	Tput	Util	Speed	Tput
-100.00	51.0	9.3	60.3	1.1435	21.8	13.1	38.4	31.1	12.0
-80.00	51.3	9.3	60.6	1.1528	21.9	13.3	40.4	29.9	12.1
-71.72	52.1	9.0	61.1	1.1711	21.1	12.9	33.6	34.7	11.7
-60.00	52.8	9.1	61.8	1.1836	21.1	13.1	33.6	35.1	11.8
-40.00	53.0	8.7	61.7	1.1807	20.7	12.8	31.4	36.6	11.5
-20.00	53.2	8.6	61.9	1.2039	20.2	12.5	28.3	39.6	11.2
0.00	48.3	8.7	57.0	1.0967	24.1	13.7	47.2	26.6	12.6
20.00	53.4	8.7	62.1	1.2296	20.0	12.4	27.7	40.3	11.1
80.00	53.3	8.4	61.7	1.2318	19.9	12.3	26.3	41.7	11.0
140.00	53.6	8.7	62.3	1.2165	19.9	12.4	27.1	41.0	11.1
200.00	51.3	7.2	58.4	1.1911	21.6	12.6	29.7	38.3	11.4

%Change	Root				Users			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	9899	17.00	7.64	24.65	9003	20.92	7.88	28.80
-80.00	9912	17.34	7.86	25.20	9075	21.16	7.65	28.81
-71.72	10122	15.77	7.72	23.49	9262	20.31	7.72	28.03
-60.00	10327	15.62	7.60	23.22	9275	20.26	7.84	28.10
-40.00	10256	14.94	7.74	22.67	9400	19.99	7.74	27.73
-20.00	10396	14.38	7.68	22.07	9351	19.43	7.60	27.03
0.00	9340	19.29	8.07	27.36	8488	21.20	7.59	28.79
20.00	10337	14.15	7.68	21.83	9505	19.61	7.70	27.31
80.00	10279	13.82	7.73	21.55	9450	19.12	7.48	26.59
140.00	10327	13.91	7.68	21.59	9534	19.14	7.54	26.69
200.00	9918	13.90	7.50	21.40	9196	17.96	7.36	25.32

%Change	Swap				Total			
	Req	Sk/Rq	Tr/Rq	Bu/Rq	Req	Sk/Rq	Tr/Rq	Bu/Rq
-100.00	2614	26.83	44.89	71.72	21516	19.84	12.27	32.10
-80.00	2762	30.74	47.37	78.11	21749	20.63	12.79	33.43
-71.72	1613	25.12	41.52	66.65	20997	18.49	10.32	28.81
-60.00	1641	24.14	39.67	63.82	21243	18.31	10.18	28.49
-40.00	1061	24.62	43.22	67.84	20717	17.73	9.56	27.28
-20.00	398	25.83	40.55	66.38	20145	16.95	8.29	25.25
0.00	4773	28.60	44.45	73.05	22601	21.97	15.57	37.55
20.00	199	23.62	39.20	62.81	20041	16.83	8.01	24.84
80.00	8	20.00	42.50	62.50	19737	16.36	7.62	23.98
140.00	179	26.59	34.86	61.45	20040	16.51	7.85	24.37
200.00	1361	23.41	42.34	65.75	20475	16.36	9.75	26.11

The following observations are made.

1. As the mean think time increases, useful throughput decreases as expected.
2. As the mean think time increases, the cpu service rate increases, and the service rate for the disk decreases. These sensitivities are significant but unexpected.
3. An X% increase in the mean think time corresponds to roughly a 0.5 times X% decrease in useful throughput.

- Useful throughput is correlated the the moments of the distribution. Generally, as the first moment increases, performance decreases. As the second moment increases, performance increases. As the third moment increases, performance decreases. These correlations are fairly significant.

#### 4. SUMMARY

The most significant findings from this experimental sensitivity analysis are:

- The big process cutoff significantly affects performance. Swapping can easily be kept within suitable tolerances by changing its value.
- Device speeds are directly correlated to parameters such as the big process cutoff, the cpu demand, the MPL, the amount of available main memory, and the user think times. This is true for the cpu speeds, and mean disk seek and transfer times per I/O request.
- The CV of the distributions of the above parameters is directly correlated with performance. This correlation is significant.

These results show that parameter interdependencies exist in actual computer systems. These interdependencies can produce significant effects on performance. Conventional models typically ignore these effects and as a result often lack accuracy and flexibility. By incorporating the interdependencies suggested by the experimental results, better analytical models can be constructed.

Total				Swap				Recharge			
Big	Small	Big	Small	Big	Small	Big	Small	Big	Small	Big	Small
32.10	12.37	12.84	21218	71.73	41.89	59.88	2914	2914	2914	100.00	100.00
33.43	13.79	20.68	21449	78.11	47.87	60.74	3703	3703	3703	80.00	80.00
38.61	10.33	18.40	20907	68.68	41.83	55.13	1813	1813	1813	-71.73	-71.73
28.49	10.18	18.41	21343	63.83	59.87	34.14	1041	1041	1041	-61.00	-61.00
31.38	9.58	17.73	20717	67.84	43.33	24.82	1041	1041	1041	-41.00	-41.00
33.38	8.39	19.88	20443	68.88	40.73	23.83	808	808	808	-30.00	-30.00
37.88	13.87	21.97	23801	73.03	44.43	23.00	4773	4773	4773	0.00	0.00
24.84	8.01	18.83	20041	63.81	39.20	23.82	108	108	108	20.00	20.00
23.88	7.83	18.38	19787	63.69	42.00	20.00	8	8	8	80.00	80.00
24.87	7.88	18.81	20080	61.43	34.88	20.89	179	179	179	140.00	140.00
20.11	8.78	16.38	20478	68.75	42.34	23.41	1981	1981	1981	200.00	200.00

The following observations are made:

- As the mean think time increases, useful throughput decreases as expected.
- As the mean think time increases, the cpu service rate increases, and the service rate for the disk decreases. These sensitivities are significant but unexpected.
- An X% increase in the mean think time corresponds to roughly a 0.5 times X% decrease in useful throughput.

# An Experimental Assessment of Resource Queue Lengths as Load Indices †

Songnian Zhou

Computer Systems Research Group  
Computer Science Division, EECS  
University of California, Berkeley  
zhou@ucbarpa.berkeley.edu

## ABSTRACT

Load indices that accurately reflect the current loads at computer system resources are crucial to the success of any dynamic load balancing scheme. However, few load indices have been experimentally validated as being suitable for load balancing. We conduct such a validation study for the resource queue lengths. We find that the average CPU and disk queue lengths during job execution have very high correlation to the response times of CPU and I/O bound jobs, respectively. However, for the type of system that we studied, the system load changes very rapidly, making the response time high unpredictable. Simulation results suggest that load balancing will drastically reduce load fluctuation. The CPU is by far the most heavily used resource in the systems we studied. While this is also true in many other environments, measurement studies are called for before reaching such a conclusion in other systems.

## 1. Introduction

As distributed computer systems become increasingly popular, the significance of load balancing is also being more widely recognized. Load balancing attempts to improve system performance by redistributing the workload submitted to the system by the users. It has been demonstrated, using analytic and simulation models, that load balancing can yield significant performance gains without requiring upgrading the system by adding more hardware [Livny and Melman 1982; Eager et al. 1986; Zhou 1986]. A prerequisite for load balancing is the availability of system load indices that properly characterize the current system loading condition. Such load indices are used by load balancing mechanisms to make job placement decisions. Although a large number of load indices have been proposed and used by the researchers in this field, few of them have been validated experimentally as being appropriate for load balancing purposes. In this paper, we conduct such a validation study for a particular load index using measurements of production system loads.

A very important system performance metric is the average response time of the jobs submitted by the users. The minimization of the average response time is often the goal of load balancing, especially for interactive computing environments. Intuitively, the

† This work was partially sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871, monitored by the Space and Naval Warfare Systems Command under contract No. N00039-84-C-0089, and by the National Science Foundation under Grant DMC-8503575. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

response time of a job is dependent on the *load* of the host on which it runs; the higher the load, the longer the response time. Therefore, a load index should ideally have a monotonic relationship with the job response time. The execution of a job requires certain amounts of various types of resources; among them are CPU cycles, disk bandwidth, memory space, and communication bandwidth. The response time of a job is determined by (1) the amounts of service time the job needs at the resources, (2) the amounts of waiting time the job spends at the resources due to contention, and (3) the synchronization and communication delays, such as locking, signals, and waiting for a message to arrive. Since (3) usually has a large variance, and is highly unpredictable, we decide not to study it here, but rather concentrate on the first two factors, which are related to the job's resource needs and the level of resource contention, respectively.

Ferrari pointed out, using mean value analysis of closed queueing networks, that a linear combination of the queue lengths of various resources should be a good predictor of the job response time, provided that the system is in steady state and that the queueing disciplines of the resources are FCFS, PS, or LCFSPR [Ferrari 1985, Ferrari and Zhou 1986]. However, these assumptions are not generally satisfied in real world computer systems. For example, in Berkeley UNIX<sup>†</sup>, a multi-class priority scheduling algorithm with round-robin preemption is used for the CPU, with the process priorities decreasing while running, and increasing while waiting to run. Jobs that return from disk I/O are given high priority in the hope that they will go back to the disk soon. Since scheduling discipline for the disks in Berkeley UNIX is one-way scan, an incoming I/O request may find itself right ahead of the disk arm, and thus be serviced first. With the complicated queueing disciplines adopted for various resources, it is not clear whether a strong linear relation exists between the resource queue lengths and the job response time. The closed system and steady state assumptions in Ferrari's model are also not generally satisfied in computer systems; users log on and off, and the number of jobs submitted can fluctuate widely. On the other hand, modeling experiences have repeatedly demonstrated that fairly simple models can provide very good results, even though some of the assumptions of the model do not strictly hold.

We conducted a series of measurement experiments on production systems under both their natural workloads and artificial workloads, with the following three objectives:

- (1) to evaluate the suitability of the resource queue lengths as load indices, and to study the sensitivity of different types of jobs to the queue lengths;
- (2) to investigate the level of contention for various types of resources in a computer system, in order to assess their relative importance;
- (3) to determine the rate at which the system load changes, and thus the effectiveness of load prediction.

In the next section, we describe the experimental set-up. In Section 3, we discuss the system load characterization in order to identify the contended resources, those causing queueing delays to jobs. The relationship between the queue lengths at the CPU and the disks, and the response times of CPU and I/O bound jobs are studied in Section 4. We study the load change rate in Section 5. Finally, we summarize our experimental findings in Section 6.

## 2. The Experimental Set-up

The first requirement of our measurement experiments was the availability of the resource queue lengths. Their values should be updated with a high frequency in order to reduce the random sampling error. Several resource queues were identified, and their instantaneous queue lengths (number of processes waiting or being served) were made available by modifying the UNIX kernel. The queues were:

<sup>†</sup> UNIX is a trademark of AT&T Bell Laboratories

- (1) **CPU ready queue** (*cpu*): the number of processes that are loaded and waiting to be scheduled to run or running. The process in execution is included.
- (2) **Disk process queue** (*dev*): the number of processes that are residing in the disk subsystem, waiting for their disk transfers to be processed. This is the aggregate queue for all the disks attached to the host. It was possible to get the queue lengths of the individual disks, but more system overhead would be incurred, and the knowledge of which disk a particular I/O operation is directed to would be required, making analyses of the measurement data more complicated. We decided to concentrate on the disk subsystem as a whole.
- (3) **Disk request queue** (*tdev*): the number of disk I/O requests that are waiting to be or being processed. The processes that initiated them may or may not be waiting (synchronous or asynchronous I/O, respectively), thus this queue is a superset of the disk process queue, and represents the actual load at the disks.

The instantaneous queue lengths were sampled every 10 milliseconds in the clock interrupt routine of the UNIX kernel, and used to compute the average queue length over a second (100 samples). We feel that this should provide enough accuracy, and that we do not need to get more than one average value per second. To reduce the overhead of tracing the system, in-kernel buffering of the queue lengths was used, so that the tracing program needed only to extract the values from the kernel at longer intervals (for example, once a minute). The overhead of system tracing, in terms of CPU time, was found to be below 0.5%.

There are other types of resources for which queue lengths do not exist explicitly, or are not appropriate as load indices. For example, a process occupies some memory space when running on the CPU, and, in addition, some memory buffer space while doing I/O. We decided to use host-wide variables such as the number of free memory page frames, and the paging and swapping rates as indices for memory contention. The network is another example of resource for which queue length is not appropriate. Processes may be waiting at a communication socket for messages to come, but not because of network bandwidth contention. Since the utilization of networks is usually low [Lazowska et al. 1985], and the protocol overhead for communication is already accounted for by the CPU queue length, we decided not to consider the network explicitly.

Most of our measurements were conducted on a production machine, *Ucbarpa*, which is a VAX-11/780 running Berkeley UNIX 4.3BSD, with 4 MB of main memory and 3 disk drives. The workload on it is usually quite heavy during the day, with load average<sup>†</sup> ranging from 3 to 10. We traced the system load indices discussed above for a number of days, with one eight-hour session per day. A number of other system load metrics, such as the one-minute load average, each individual disk's transfer rate and utilization, the multiprogramming level, and the percentages of CPU time spent executing system kernel code and user code, were also collected.

With the natural workload of the system as our background load, we ran benchmark jobs periodically and measured their response times (*r*). The data collected enabled us to conduct a series of regression studies to evaluate the potential of the resource queue lengths for predicting the job response times.

Three types of jobs were used at different times.

- (1) A *troff* job (TROFF) that is mostly CPU bound, with some I/O. Two input text file sizes were used. We chose *troff* jobs as benchmarks because they are often big and therefore good candidates for load balancing
- (2) A purely CPU bound benchmark (CPU) that performs a lot of arithmetic computations.

<sup>†</sup> The load average in UNIX is an exponentially smoothed average of the total number of processes in the CPU queue and the disk queues, and is widely used as a system load index

- (3) A predominantly I/O bound benchmark (IO) that copies a large file block by block in reverse order to defeat the read ahead mechanism in UNIX. Since the size of the file is chosen to be larger than the size of the system disk cache, which uses an LRU replacement algorithm, all I/O operations are forced to the disk.

To minimize the effect of the extra workload introduced by the benchmarks, we only ran one type of benchmark for each session. Each of the benchmarks consumed 4-10% of the total CPU time.

### 3. System Load Characterization

From the measurements of the system queue lengths and other variables, we were able to study the system's load characteristics. Below is a summary of our findings.

#### CPU:

The CPU is usually heavily contended. Its queue length was observed to be above 3 during the day, and the utilization at or near 100% (no or little idle time).

#### Disks:

They are usually lightly loaded. The average queue length at the busiest disk is around 0.3 when the system load average is 5-10. The corresponding disk utilization (percentage of time the disk is busy) is about 25%. This can be explained by the effectiveness of the large disk cache, which is reported to have an average block hit ratio of 85% [Leffler et al. 1984]. Figures 1 and 2 show the distributions of the utilization and transfer rate for the most heavily used disk observed during a day of heavy system load.

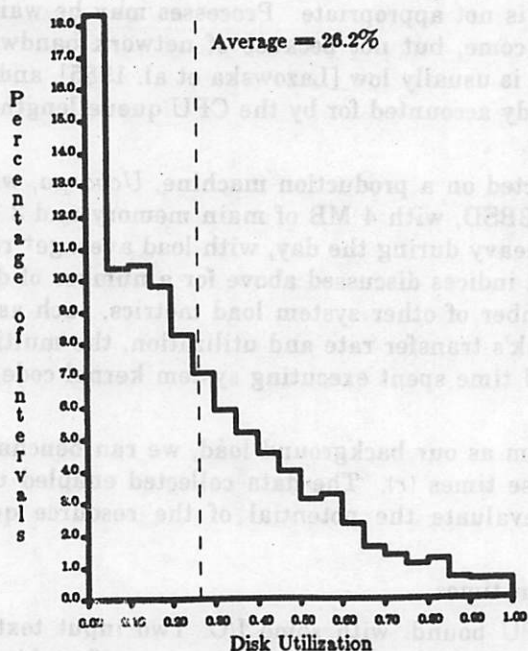


Figure 1. Disk Utilization Distribution.  
(measured in 5 second intervals)

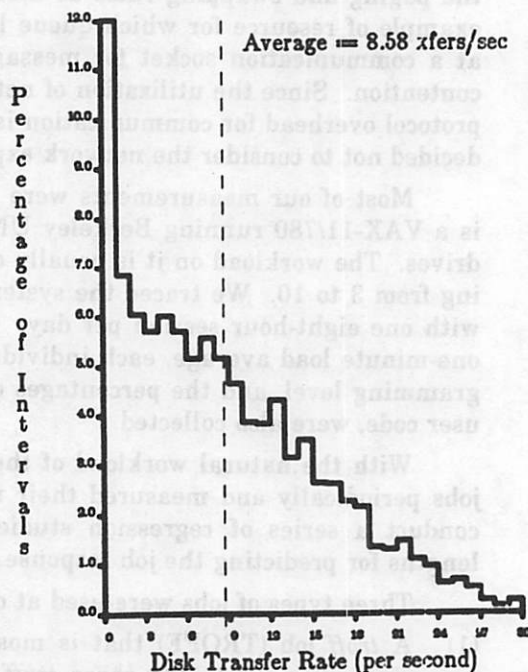


Figure 2. Disk Transfer Rate Distribution  
(measured in 5 second intervals)

## Memory:

There is usually plenty of memory, thus few pages that are currently being used are paged out, and few forced process swap-outs occur. The distributions of the number of free memory pages and the paging rates for a day of heavy load are shown in Figures 3 and 4.

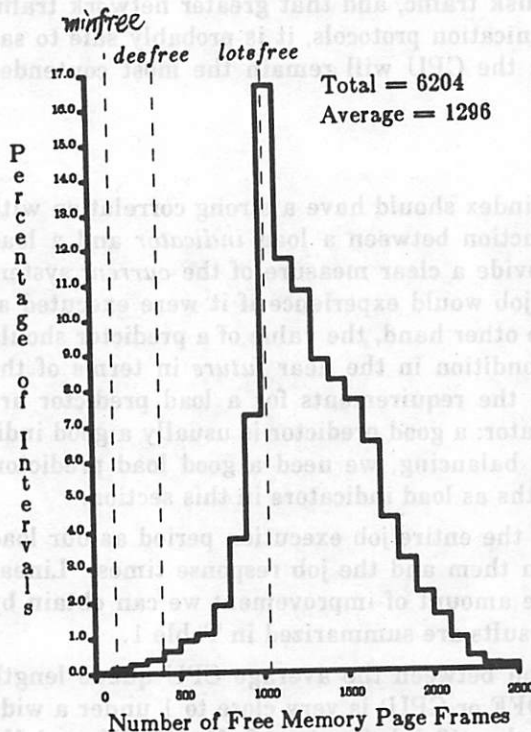


Figure 3. Distribution of Free Memory Page Frames (measured in 5 second intervals)

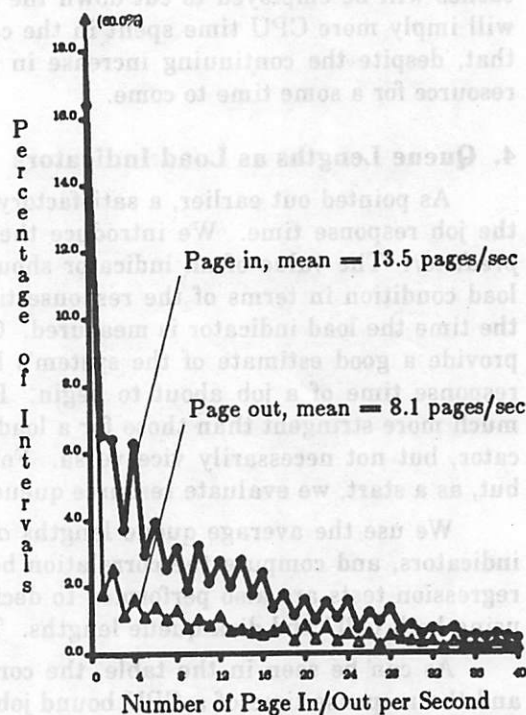


Figure 4. Paging Rates (measured in 5 second intervals)

There are three important parameters in the memory management system of Berkeley UNIX. If the number of free main memory page frames is above *lotsfree*, the page daemon is not activated, thus no page-outs occur. This was the case 80% of the time on the day the data in Figures 3 and 4 were collected. If the number of free memory pages falls below *lotsfree*, the page daemon will scan the page frames and free up those that have not been used for a while. Processes will be swapped out only if the number of free page frames falls below a very low level, *minfree*, the average number of free frames over a recent time interval is below a desirable level, *desfree*, and the CPU load is very high. In that case, only processes that have been dormant for quite a while are usually swapped out. We observed a swapping rate of about 4 processes per minute during the measurement session.

From the above observations, we conclude that the CPU is the bottleneck of the system. This is not peculiar to the machine we measured or to Berkeley UNIX running on a VAX. Our study of the diskless SUN workstations also shown that, although all the paging and swapping, in addition to the file I/O operations, are performed through the network on the shared file servers, the workstation and file server CPU's are still the most heavily contended resources. The average number of processes on the workstation waiting for I/O rarely exceed 0.2. Similar observations have been reported for other systems too. For example, Lazowska and his coworkers found that, in distributed systems based on diskless workstations running SUN's version of Berkeley UNIX, the V system, and the Domain

software of Apollo computers, the utilizations of the file server CPU's are two or more times higher than those of the disks and the networks [Lazowska et al. 1985]. Therefore, the CPU queue length can be expected to be an important load index. In Ferrari's linear combination of queue lengths, the CPU term will tend to be the dominant one, except for the extremely I/O bound jobs. This is confirmed by the analyses presented in the next sections. Considering that the memory size will increase very rapidly, hence larger and larger disk caches will be employed to cut down the slow disk traffic, and that greater network traffic will imply more CPU time spent in the communication protocols, it is probably safe to say that, despite the continuing increase in speed, the CPU will remain the most contended resource for a some time to come.

#### 4. Queue Lengths as Load Indicators

As pointed out earlier, a satisfactory load index should have a strong correlation with the job response time. We introduce the distinction between a load *indicator* and a load *predictor*. The value of an indicator should provide a clear measure of the *current* system load condition in terms of the response time a job would experience if it were executed at the time the load indicator is measured. On the other hand, the value of a predictor should provide a good estimate of the system's load condition in the near *future* in terms of the response time of a job about to begin. Hence, the requirements for a load predictor are much more stringent than those for a load indicator: a good predictor is usually a good indicator, but not necessarily vice versa. For load balancing, we need a good load predictor, but, as a start, we evaluate resource queue lengths as load indicators in this section.

We use the average queue lengths *during* the entire job execution period as our load indicators, and compute the correlation between them and the job response times. Linear regression tests are also performed to decide the amount of improvement we can obtain by using both CPU and disk queue lengths. The results are summarized in Table 1.

As can be seen in the table, the correlation between the average CPU queue length and the response time of a CPU bound job (TROFF or CPU) is very close to 1 under a wide spectrum of system load conditions, both live and artificial (Sessions I, II, III, IV, and V). On the other hand, the correlation between the aggregate disk queue length and the response time is much lower, and the amount of improvement obtained by taking it into consideration is negligible (the correlations in the regression tests are only slightly better than that with CPU queue length alone). This is not only true for live workloads that usually do not cause much contention on the disks, but also with artificial workloads having high disk demands. It should be noted that, with the live workloads, only the aggregate disk queue length is available, and the correlation between it and the job response time can be expected to be lower than that when the queue length of the disk to which the job's I/O is directed is used. With artificial workloads, however, a heavy traffic to/from a particular disk is generated, while all the other disks are virtually idle. Consequently, the aggregate disk queue coincided with the single disk queue, yet, we still observed a low correlation. We can safely conclude that the average CPU queue length alone is an excellent load indicator for CPU bound jobs. This finding is intuitive, yet very reassuring.

The situation with I/O bound jobs is slightly more complicated. Under a live workload, since the disks are not heavily loaded, the response time of an I/O bound job is found to be insensitive to both the CPU and the disk queue lengths. To study the dependency of the I/O bound job's response time upon disk queue length, we were forced to use an artificial workload that generated a heavy traffic to/from a particular disk. Results similar to the CPU bound jobs were obtained: when the disk is contended for ( $t_{dev} > 2$ ), the correlation between the disk queue length and the response time is very high, whereas the CPU queue length has a much lower correlation (Sessions VI and VII). Considering both CPU and disk queues provides little improvement.

The above discussion can be summarized as follows: a good load indicator is job-type-dependent; for CPU bound jobs, the CPU queue length is predominant, whereas for I/O

Table 1. The Correlation between Job Response Time and Queue Lengths

Session	I	II	III	IV	V	VI	VII
Job Type	TROFF	TROFF	TROFF	CPU	CPU	IO	IO
Load Type	<i>live</i>	<i>live</i>	<i>live</i>	<i>live</i>	<i>art,mix</i>	<i>art,io</i>	<i>art,mix</i>
sample size	109	104	188	52	22	53	101
$cor(cpu,r)$	0.9699	0.9767	0.9528	0.9505	0.9838	0.7459	0.0666
$cor(dev,r)$	0.4873	0.8179	0.7401	0.5739	0.4758	0.9429	0.8538
$cor(tdev,r)$	0.5097	0.8388	0.6582	0.6354	0.4686	0.9346	0.8412
$reg(cpu,dev;r)$	0.9702	0.9795	0.9604	0.9505	0.9918	0.9441	0.8611
$reg(cpu,tdev;r)$	0.9703	0.9794	0.9588	0.9507	0.9927	0.9346	0.8597
$cor(cpu + dev,r)$	0.9620	0.9793	0.9385	0.9446	0.9754	0.9394	0.3897
$cor(cpu + tdev,r)$	0.9537	0.9788	0.9564	0.9430	0.9618	0.9333	0.4547
$mean(cpu)$	3.3028	4.1186	3.0932	2.2770	4.2539	0.3606	5.4651
$mean(dev)$	0.4930	0.4501	0.4654	0.4574	3.3939	2.0377	3.8015
$mean(tdev)$	0.9852	0.8317	0.9029	0.7431	4.2289	2.5192	4.7390
$mean(r)$	101.11	136.63	44.36	201.5	897.8	97.69	119.04
$stdv(r)$	74.48	82.64	29.44	98.6	572.7	20.13	31.95

- $cor(a,b)$  - correlation coefficient between  $a$  and  $b$ ;  
 $reg(a,b; c)$  - correlation coefficient obtained by doing linear regression on  $c$  using  $a$  and  $b$ ;  
 $r$  - job response time;  
 $mean$  - mean queue length during the whole measurement session, as a measure of the average load;  
 $stdv$  - standard deviation of the queue length distribution during the whole measurement session, as a measure of the load fluctuation;  
 $live$  - session under live, production workload;  
 $art$  - session under artificial workload;  
 $mix$  - artificial workload consisting of a mixture of CPU and I/O bound jobs;  
 $io$  - artificial workload consisting of only I/O bound jobs.

bound jobs, if there is high contention at the disk (which was rarely observed), the disk queue length is sufficient; otherwise, the job response time is insensitive to system load.

We have demonstrated experimentally that the resource queue lengths, if used properly, can be very good load indicators. Whether they are also good load predictors is determined by the rate at which workload fluctuates, in other words, by whether Ferrari's steady state assumption holds well in computer systems. We study this problem in the next section.

## 5. Queue Lengths as Load Predictors

To evaluate the ability of the resource queue lengths in predicting future jobs' response times, we used the average queue lengths for a period (15, 30 or 60 seconds) *before* the start of the job, instead of those *during* the execution of the job. Table 2 shows the various correlations for Sessions I, II and IV listed in Table 1.

Table 2. Queue Lengths as Predictors of Response Time.

	PAST15	PAST30	PAST60
Session I <i>compare: cor(la, r) = 0.6964</i>			
<i>cor(cpu, r)</i>	0.6915	0.6928	0.6353
<i>cor(dev, r)</i>	0.1948	0.2932	0.2906
<i>cor(tdev, r)</i>	0.1639	0.3137	0.3210
Session II <i>compare: cor(la, r) = 0.3509</i>			
<i>cor(cpu, r)</i>	-	0.4763	-
<i>cor(dev, r)</i>	-	0.5838	-
<i>cor(tdev, r)</i>	-	0.6155	-
Session IV <i>compare: cor(la, r) = 0.6831</i>			
<i>cor(cpu, r)</i>	0.6866	0.7003	0.6867
<i>cor(dev, r)</i>	0.4130	0.4725	0.5076
<i>cor(tdev, r)</i>	0.3376	0.4522	0.4689

PAST15, PAST30, PAST60: average queue length for the 15, 30 or 60 seconds immediately before the job starts

The correlation coefficients between queue lengths and response times are much lower than in the "during" case, almost never exceeding 0.7. Prediction using average queue length is not much better than using the value of the load average (*la*) immediately before the job starts (see *compare*). We attempted to improve the prediction by using an exponential smoothing algorithm, but the improvement was very small. Furthermore, the higher the load, the greater the fluctuation observed, and the poorer the prediction (Session II). The consistently poor correlation in a number of sessions seems to suggest that there is a "sound barrier" that defies any attempt to predict the future load. The resource queue lengths are changing so rapidly that their values before the job starts poorly predict those during the job execution, and the job's response time cannot therefore be accurately predicted. In other words, the steady state assumption in Ferrari's model does not hold. This is confirmed by our measurements of the rate of change of the CPU queue length. Figure 5 shows the distribution of net CPU queue length changes during 30-second intervals over an entire session (about eight hours). † 33% of the time, the net change is 3 or beyond. For an average queue length of 6, which is fairly high, this represents a 50% change in CPU load. For the machine we measured, the average net change in CPU queue length in a 30-second interval is 2.31.

The wide and rapid changes in system load make predicting future loads quite difficult, thus load balancing policies based on queue lengths are bound to make mistakes. The prediction of job response times gets worse as the job gets longer, which is bad for load balancing because load balancing policies are mostly concerned with long jobs, due to the non-zero costs of moving jobs over the network. However, it should be pointed out that, for load balancing, we are interested in *comparing* the loads of a number of hosts in order to

† To make the measurements here compatible with those from the simulation to be discussed shortly, we use the instantaneous CPU queue length, rather than the average queue length over one second, which has been used up to this point.

choose one that is lightly loaded, rather than in determining the absolute response times of the job if it were executed on each of the eligible hosts. Another factor that is likely to ease the difficulty of prediction is that load balancing, if effective, should tend to reduce the fluctuations of the load on each machine, thereby improving the accuracy of response time prediction. In other words, load balancing should provide a responsive *negative feed-back* that will tend to stabilize machine loads. This conjecture was confirmed by the results obtained from a trace-driven simulator that we constructed for studying load balancing [Zhou 1986]. Records containing job arrival times and resource consumptions for all the commands submitted to a production system during a number of tracing sessions were used to drive a model that simulated the executions of the commands on a number of hosts, with and without load balancing. It was found that load balancing, besides effectively decreasing the average command response time, also tends to smooth out the temporal fluctuations in the load of each machine. For a typical machine, the distributions of load changes during 30-second intervals as described above were computed for cases with and without load balancing, and are shown in Figure 6. The average net change in load decreased from 2.42 for the no-load-balancing case to 0.93 for that with load balancing.

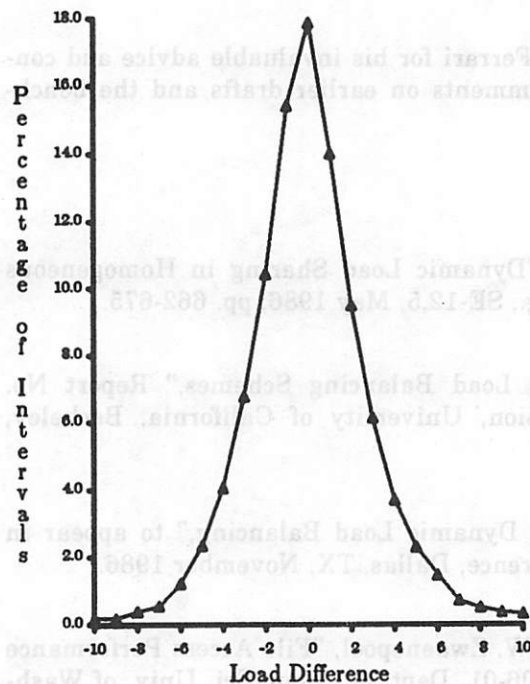


Figure 5. Load Change Frequency (in 30 Seconds) from Measurement

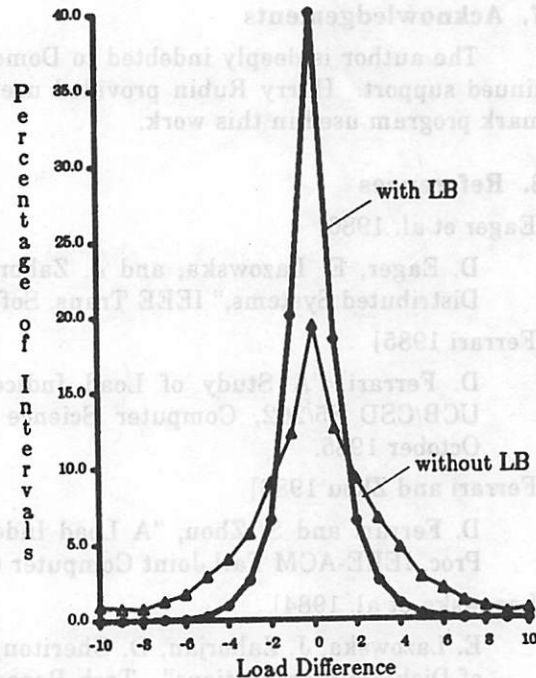


Figure 6. Load Change Frequencies (in 30 Seconds) from Simulation

## 6. Conclusions

The CPU is the most contended resource in systems similar to the one we measured, while there is usually plenty of main memory, so that little paging out and few forced process swap-outs occur. Traffic to and from the disks is light, and the queueing times processes spend at the disks are low. The addition of more disk drives is usually because of the need for more on-line storage space, rather than of constraints on disk bandwidths. This is also true for many other systems, and is going to be true for some time to come.

The resource queue lengths can be excellent load indicators if used properly. For CPU bound jobs, the CPU queue length is closely related to the response time, whereas the disk queue lengths have much lower correlations to it. For I/O bound jobs, the particular disk queue length reflects the response time very well when processes are queued up there. However, for live workloads characterized by short queues at the disks, the response time is insensitive to both CPU and disk queue lengths.

The load of live systems changes very rapidly, making response time predictions difficult. On the other hand, load balancing is expected to smooth out the wide fluctuations, thereby improving the predictions. This conjecture is confirmed by simulation studies of load balancing.

We have used load indices based on resource queue lengths in both trace-driven simulation and implementation of load balancing. A reduction in the mean job response time of 30-50% were observed in both cases [Zhou 1986]. While these results are not proof that an appropriate linear combination of resource queue lengths is the best load index, we feel that such a load index is very satisfactory in most cases.

## 7. Acknowledgements

The author is deeply indebted to Domenico Ferrari for his invaluable advice and continued support. Harry Rubin provided useful comments on earlier drafts and the benchmark program used in this work.

## 8. References

[Eager et al. 1986]

D. Eager, E. Lazowska, and J. Zahorjan, "Dynamic Load Sharing in Homogeneous Distributed Systems," IEEE Trans. Soft. Eng., SE-12,5, May 1986, pp. 662-675.

[Ferrari 1985]

D. Ferrari, "A Study of Load Indices for Load Balancing Schemes," Report No. UCB/CSD 85/262, Computer Science Division, University of California, Berkeley, October 1985.

[Ferrari and Zhou 1986]

D. Ferrari and S. Zhou, "A Load Index for Dynamic Load Balancing," to appear in Proc. IEEE-ACM Fall Joint Computer Conference, Dallas, TX, November 1986.

[Lazowska et al. 1984]

E. Lazowska, J. Zahorjan, D. Cheriton, and W. Zwaenepoel, "File Access Performance of Diskless Workstations". Tech Report 84-06-01, Dept. of Comp. Sci, Univ. of Washington, June 1984.

[Leffler et al. 1984]

S. Leffler, M. Karels, and K. McKusick, "Measuring and Improving the Performance of 4.2 BSD," Proceedings of the Salt Lake City Usenix Conference, pp. 237-252, June 1984.

[Livny and Melman 1982]

M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," Proc. ACM Computer Network Performance Symposium, April 1982,

[Zhou 1986]

S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," Tech. Report, UCB/CSD 87/305, Computer Science Division, University of California, Berkeley, September 1986.

## A Parallel Programming Process Model

Bob Beck (*sequent!rbk*)

Dave Olien (*sequent!dmo*)

Sequent Computer Systems  
15450 Southwest Koll Parkway  
Beaverton, Oregon 97006  
(503)626-5700

### ABSTRACT

Parallel computers are now readily available in the marketplace. The UNIX execution environment matches these machines naturally, due to the ease with which multiple processes may be used and the ability to connect them via pipes and other mechanisms. If the underlying software supports dynamic load balancing of processes across processors, large gains in system throughput can result. Another benefit of parallel computers is the ability to apply multiple processors concurrently working towards the solution of a problem, greatly increasing the speed of computation. Arranging this sort of computation is commonly called *parallel programming*.

The standard UNIX process model assumes multiple isolated processes. UNIX implementations typically go to great length to insure processes cannot interfere with each other's computation. This model must be extended to properly take advantage of the power available in parallel computers. This paper presents a process model for running parallel programs on UNIX in a shared-memory multiprocessor. The need for extensions to the standard UNIX process model is motivated, and a simple set of extensions to the process model, standard languages, and run-time support are presented, including some of the necessary kernel support. This model has proven to be quite easy to use, flexible, and powerful; several examples of its use are given.

December 4, 1986

## A Parallel Programming Process Model

*Bob Beck (sequent!rbk)*

*Dave Olien (sequent!dmo)*

Sequent Computer Systems  
15450 Southwest Koll Parkway  
Beaverton, Oregon 97006  
(503)626-5700

### 1. Introduction

Parallel computers are now readily available in the marketplace. They come in a variety of architectural forms, roughly ranging from loosely-coupled distributed memory machines such as a hypercube [Seitz 1985], to more classic shared-memory multiprocessors [Fielland 1984], with numerous variants to fill the spectrum. They are called *parallel* computers because they all contain more than one actual processing unit, and have the potential to apply multiple processors concurrently working towards the solution of one or several problems.

#### 1.1. Multiprogramming

One problem well suited to certain parallel computers is that of running multiple concurrent but largely unrelated processes, typically called *multiprogramming*. The UNIX<sup>1</sup> operating system encourages multiprogramming by making it easy to create concurrent processes, and by providing mechanisms to connect them in useful ways (e.g., process *pipe lines*). A typical UNIX system runs a time-sharing mix of processes, where frequently more than one process (often many more) is ready to execute at a given time. UNIX implemented on a single-processor computer executes these processes in apparent concurrency; only one process runs at a time, and the processor resource is *time-sliced* among runnable processes so that all make progress. It is possible to implement UNIX on a shared-memory multiprocessor in a manner that manages the processors transparently to the application by dynamically balancing the processing load among the processors [Beck 1985]. Shared memory allows a processor to execute any process without first copying large portions of that process's memory into a local processor memory. Shared memory also makes communication and sharing of information among processes simple and efficient. Such a system executes several of the ready processes in actual concurrency instead of apparent concurrency, resulting in an increase in overall system throughput. This throughput increase can be dramatic [Rodgers 1985]. Furthermore, since the operating system manages the processes in a manner which is transparent to applications, these throughput increases can be realized with little or no modification to the applications. It is required only that the applications be already implemented using several processes (for example, a pipe construct) or that several instances of the application can be run concurrently (such as the compilation of several source files, or a multi-user workload).

#### 1.2. Parallel Programming

A machine that can execute unrelated processes *concurrently* also has the potential to execute **tightly-coupled** processes concurrently, to achieve an increase in computation speed for a single application. On monoprocesor systems, these applications typically consist of a single program which is executed by a single UNIX process. Such a program can be modified so that on a multiprocessor, its computation can be executed concurrently by several tightly-coupled processes, each running on a

<sup>1</sup> UNIX is a registered trademark of AT&T.

separate processor. Such an application is said to be running *in parallel*. These tightly-coupled processes often need to exchange information or synchronize with each other very frequently. Traditional UNIX inter-process communication schemes, such as pipes or sockets, typically are not adequate for these communication needs. Instead, shared memory can be used as the basis for inter-process communication.

Programming an application to run in parallel is commonly called **parallel programming**. It has been shown that most programs contain exploitable parallelism to some degree, often a high degree [Nicolau 1981]. If a sufficient portion of a program can be run in parallel, the execution time of that program can be greatly reduced. In some cases speedups of a factor of 20 or more are possible, even in programs which were not originally designed with parallel execution in mind [Sequent 1986]. Programs explicitly designed for execution on parallel machines have even more potential for speedup. It is highly desirable that the power of multiprocessor computers be easily applied to applications, making these potentials for speedup into reality.

Although conceptually straightforward, the details of actually supporting the execution of parallel programs are quite involved. Even those languages that embody parallel constructs (e.g., Ada<sup>2</sup> [DOD 1983], and Concurrent C [Gehani 1985]) assume some underlying support for their basic mechanisms. There are no standards and little in the literature that seems to address such parallel program support mechanisms. This paper presents a process model for constructing and executing user-mode parallel programs in a shared-memory multiprocessor running UNIX. The model involves some simple extensions to the UNIX process model, user-level library support, and additional operating system support. Parallel programming in the C language, FORTRAN, and Ada are then discussed. An example of use is presented.

The model has evolved through several designs and implementations; it is supported in the DYNIX<sup>3</sup> operating system, the version of UNIX running on Sequent Balance machines. The current model is more powerful and flexible than earlier versions, but some interesting issues remain. Several of these issues and ideas for future support are presented.

## 2. Parallel Process Model

This section proposes a process model which supports the execution of parallel applications on a shared memory multiprocessor. A simple definition of a shared memory multiprocessor is a machine that contains some number of processors (typically more than two) connected to a common, or shared, memory via an efficient switch, often a shared bus. This architecture is illustrated by Figure 1. The important characteristics of the model are that the processors are all equivalent (there is no *master* processor), they equally share access to the memory, and all have access to IO devices.

A UNIX process may be viewed as a **virtual processor**: it has a machine state (register context), memory resources (typically virtual memory), and connections to IO devices, files, etc. The virtual processor is a higher level machine than the hardware it runs on, since it includes operating system services such as virtual memory, and file system. If the processors in Figure 1 are re-labeled as *processes*, the figure could represent a set of UNIX processes sharing memory. Each process has memory, some of which is shared with other processes; each has a machine state, and connections to files. If the number of virtual processors is at most the number of physical processors in the machine, they can all run concurrently. The result is a **virtual multiprocessor**, constructed from one or more virtual processors.

The virtual multiprocessor outlined above defines a parallel process model. This model extends a single process model in the same way a multiprocessor extends a monoprocessor hardware architecture. Just as the virtual processor is more powerful than the physical processor it runs on, the virtual multiprocessor is more powerful than the hardware it runs on. Thus any algorithm that can be executed on a multiprocessor hardware should be executable in the parallel process model. However, a

<sup>2</sup> Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

<sup>3</sup> Balance and DYNIX are registered trademarks of Sequent Computer Systems, Inc.

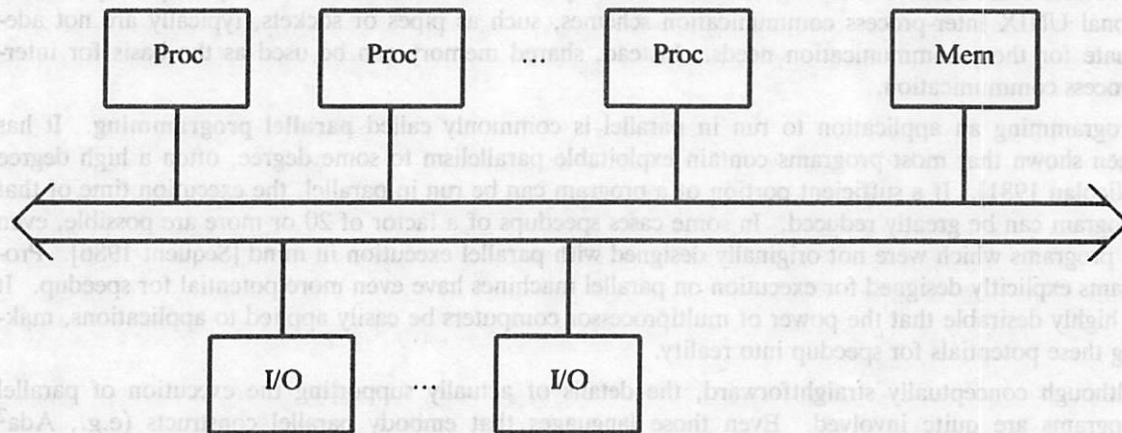


Figure 1: Shared Memory Multiprocessor

number of additional facilities are necessary to make the model truly useful:

- flexible shared memory supported by the operating system, to allow the set of related parallel processes to share appropriate parts of their address spaces. This provides a mechanism for efficient sharing and communication of data among parallel processes.
- compiler, assembler, and linker support to conveniently declare shared and non-shared memory. These facilities should make creating and referencing objects in shared memory as easy and natural for the programmer as declaring non-shared objects is in non-parallel programs.
- run-time support to manipulate shared memory as easily as UNIX processes manipulate private memory. Shared memory management routines should be extensions of UNIX routines for managing private memory. Much monoprocessor software which use the private memory facilities could be modified to use the new shared memory facilities.

The remainder of this paper discusses a parallel process model and language, run-time and operating system support for parallel programs.

### 3. Parallel Process Model

This section gives an overview of the standard, single-process UNIX programming model and discusses extensions to this model for developing parallel programs.

#### 3.1. Goals

A parallel process model should satisfy the following goals:

- Provide a model which applications can use to achieve real performance improvement over monoprocessor implementations.
- Provide basic support for shared memory parallel programs.
- Provide a sound basis for constructing a variety of higher level abstractions. Such abstractions include multitasking, microtasking, parallel Ada run-time system, and other parallel language run-time support.
- Allow existing single-process programs to be easily extended into parallel versions.
- Extend the concepts and constructs used in single-process programs, not replace them. New interfaces should be intuitive extensions of existing interfaces where possible.

- New facilities should be consistent with the UNIX style of programming. Simple things should be simple to do, harder things should be possible.
- The implementation should be efficient. It is usually the case that parallel programs have non-trivial run times, otherwise there is little reason to run in parallel. Some additional operating system and run-time overhead is acceptable, but it must be minimized.

### 3.2. Standard UNIX Process Model

The address space of a process executing on UNIX has three *segments*, called text, data, and stack. The **text segment** is the code of the program, usually implemented as read-only and shared among all processes executing a given program. The text segment is typically fixed in size and cannot be changed by the process unless it executes a different program binary.

The **data segment** is the process-local data of the program, used as a private read/write memory by the process. No other process can access or alter the private data memory of a process. The data segment contents are initially determined by the program image the process executes (the *executable binary* file). The process can alter the size of its data segment via the `brk()` system call. The data segment is composed of two parts: initialized data and uninitialized data. Only the initialized data is present in the executable binary; the uninitialized data is represented by its size only, so the system knows how large a data segment to create. In this context, *uninitialized* actually means not explicitly initialized in the executable binary file; the system initializes the memory to zeroes when first referenced. This allows an executable binary file to represent a large initial static data without committing disk storage space to the file. For historical reasons, the uninitialized data is called **bss**.

The **stack segment** is the process-local stack, typically used for subroutine linkage and automatic variable storage. Like the data segment, the stack segment is private to the process. In most UNIX implementations, the stack is automatically grown in size as the process requires it; this avoids the need to pre-specify the size of the stack, which can be difficult or impossible for many programs.

In order to allow both the data and stack of a process to grow dynamically, they are typically arranged at opposite ends of the process address space and grow towards each other. Attempting to grow the stack or data segments so that they overlap typically results in an error. The address space of a standard UNIX process is illustrated in Figure 2.

UNIX provides a set of library routines and system calls to manipulate the process memory image in convenient ways. These include support to grow or shrink the data segment (`brk()`, `sbrk()`), and a heap memory manager (`malloc()`, `free()`). These routines have simple interfaces and are easy to use; a number of higher-level mechanisms can be built from them.

In addition to its memory resources, a UNIX process may have some number of connections to files, IO devices, network connections, etc. These connections are called **file descriptors**. Most UNIX implementations allow at least 20 file descriptors per process, and allow them to reference objects such as files, pipes, sockets, and IO devices.

To protect processes from each other, UNIX uses memory management hardware to arrange that a process cannot reference another process's address space, nor the address space of the operating system. The basic assumption is that processes are mutually independent and don't wish to share portions of their address space; unrestricted sharing (for example, in a hardware environment that doesn't support memory protection) would be dangerous and lead to obscure bugs or system failures. Many types of programs run quite well on this model, evidenced by the large number of applications that run on a time-shared UNIX machine. There are usually some ways for one process to view the address space of another (e.g., `ptrace()`, `/dev/mem`), but these mechanisms tend to be awkward and not very general.

### 3.3. Parallel Process Model

The model described above can be extended in a consistent manner to support multiple processes executing a parallel program. The most basic need is to allow programs to declare which variables are

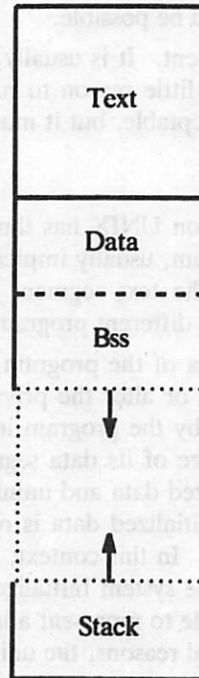


Figure 2: UNIX Process Address Space

process-private, and which are shared among the processes executing the parallel program. The `fork()` system call is the basic mechanism in UNIX for creating processes. When a process forks, a new process, called a child process, is created. The child process's memory is a copy of the original process, called the parent process. The parallel process model modifies the behavior of `fork()` to treat process-private and shared memory differently. In this context, *shared* and *private* refer to the process scope of a variable after a process forks. A **private** variable is copied on fork; with each resulting process has its own copy of that variable. Each process may write on the variable, but the value of the variable changes only in the individual process. A **shared** variable is available to both parent and child process after fork. Both processes share access to the same copy of the variable; either can modify the variable and it is implicitly modified in the other process. Thus *shared* may be viewed as a shorthand for *shared after fork*. Shared variables support a shared memory programming model; the speed at which an update of the variable is made visible to other sharing processes is that of the memory system in the machine. Of course, if a process containing a shared variable forks more than once, there will be several processes with access to the same shared variable.

The single-process address space layout is extended for multiple processes by creating analogs to initialized and uninitialized data, illustrated in Figure 3. The new address space sections are referred to as **shared data** and **shared bss**, after their private address space counterparts. The terms **private data** and **private bss** refer to the process-private data, which was simply called *data* and *bss* in the single-process model. Parallel programs need to manage private data as well as shared data; thus shared data is placed after private data in the address space, leaving room for expansion of the private data. An address space hole fills the space between shared data and the process private stack; shared data may be expanded into this hole, much as private data is in the single-process model. The stack remains process-private, to avoid having multiple processes attempting to execute on the same stack. Higher-level abstractions, such as Ada tasking or the microtasking facilities discussed later in this paper, can place stacks in shared memory if needed.

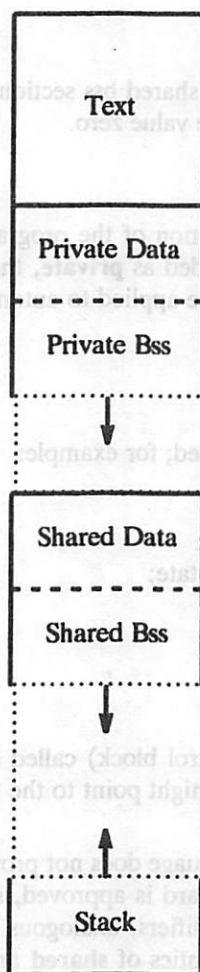


Figure 3: Parallel Program Address Space

#### 4. Language Extensions

This section discusses extensions to the C and FORTRAN languages, assembler, and linker to support the parallel process model. The primary goal of these extensions is to provide easy-to-use constructs for declaring and initializing shared-memory, yielding an intuitive model in which to write shared-memory parallel programs.

##### 4.1. C Language

The C language is extended with two new keywords: **shared** and **private**. These keywords may be used in a variable declaration to specify the process scope of the variable after fork, as described earlier. The **shared** and **private** keywords modify the storage class of the variable; after a fork the variable will be shared with other processes, or private to the process, respectively. Variables declared as **private** are placed in the private data segment in the program. **Shared** variables are placed in the shared data segment of the program. Most other attributes of C variables are available; for example, shared and private variables may still be **static** (module local) or **extern** (globally visible), and may be explicitly or implicitly initialized. For example,

```
shared int x;
```

declares an integer *x* and places it in the shared bss section of the program; that is, *x* is shared after fork, and on first reference it will have the value zero.

```
shared int x = 3;
```

Places *x* in the initialized shared data section of the program with the initial value 3. If the **shared** keyword were omitted above, or were coded as **private**, then *x* would be a process private variable. The **shared** and **private** keywords cannot be applied to automatic variables. For example,

```
shared register int z;
```

is not a valid declaration.

All forms of C type declarations can be used; for example:

```
struct tcb {
    struct tcb *tcb_next;
    int      tcb_somestate;
};
```

```
shared struct tcb sh_tcb;
private struct tcb *cur_tcb;
```

declares a shared *tcb* (short for task control block) called *sh\_tcb* and a process private pointer to a task control block called *cur\_tcb*, which might point to the task control block of the task running in a given process.

The current ANSI standard for the C language does not provide any facilities for shared memory programming. When and if the ANSI standard is approved, **shared** and **private** could be added to an ANSI standard compiler as new type modifiers, analogous to the **const** and **volatile** keywords in the proposed standard. The syntax and semantics of **shared** and **private** in the DYNIX C compiler are not consistent with the proposed ANSI standard for the C language. Instead, a simpler storage class modifier syntax has been implemented, since the DYNIX C compiler doesn't implement the full language specified in the proposed standard. Until the ANSI standard is approved, the current extensions suffice. For more detail on the proposed ANSI standard for C, see [CSTD 1985].

#### 4.2. Assembly Language

The DYNIX assembler accepts a single new pseudo-operation, **.shdata**. The **.shdata** pseudo-operation is analogous to the **.text** and **.data** pseudo-operations present in most UNIX assemblers. When **.shdata** is in effect, all initialized variables are placed in initialized shared data and **.comm** and **.lcomm** directives place variables in shared bss (with external or static module scoping, respectively). This extension is sufficient to allow the C compiler to produce code for the shared and private declarations outlined above.

#### 4.3. Object Format

The object file format is extended by adding fields to the header to describe the sizes of the initialized and uninitialized shared data sections of the file, and placing the initialized shared data after the initialized private data in the file. The resulting object file format is very similar to the 4.2bsd *a.out* format upon which it is based. The symbol table types are extended to distinguish between private and shared symbols.

#### 4.4. Loader

Loader has been extended to support the parallel process model in the following ways:

- The loader understands the shared parts of the object files and produce an executable object file that collects the private and shared parts of the input files separately.
- The loader provides error checking by complaining if the same symbol has shared process scope in one module and private in another, to help find scoping bugs before actually running the program.
- The loader creates a hole between the end of private data and the start of shared data to allow dynamic expansion of private data. The default size of this hole is chosen to be sufficient for the private data expansion needs in most programs; it may be altered with a loader invocation switch.
- A new invocation switch allows arbitrary *.comm* (common) symbols to be declared *shared* instead of *private*. This is used to force data structures to reside in shared-memory without modifying the source code of the program.

#### 4.5. FORTRAN

FORTTRAN provides poor data typing and abstraction facilities. The primary mechanism FORTRAN supplies for gathering variables is the **common block**. FORTRAN programmers are accustomed to gathering related variables into named common blocks for a variety of reasons. The new loader switch is used to place named common blocks in shared-memory. Thus no changes to the FORTRAN compiler were necessary to provide shared-memory to FORTRAN programs. This is somewhat crude, but experience has shown the mechanism is sufficient for most applications [Sequent2 1986].

### 5. Run-Time Library Support

The following sections outline the run-time support facilities for developing parallel programs. These facilities include shared memory initialization, support for dynamic growth of shared memory, and shared memory heap management. Where there is no reason to change interfaces, they were brought forward intact.

#### 5.1. Run-time Initialization

The UNIX single-process programming model requires no explicit initialization code. Any necessary initializations are done before calling the *main* program, or implicitly on first call of individual functions. For example, the C language run-time start code reformats the argument data UNIX places on the stack into the form for the *argc*, *argv*, and *environ* arguments of the *main* procedure. Parallel programs also need no explicit run-time initialization. All private and shared regions are set up before the *main* program is started. The parallel program need only fork to start its parallel activity.

The parallel programming library implements shared memory using the DYNIX kernel's mapped file facilities. The DYNIX *mmap()* system call can be used to map portions of a file into an arbitrary location in a process's address space (see section 6.1). If two or more processes map the same portion of the same file, those processes are effectively sharing memory. The run-time support library performs initialization before calling *main()* in the parallel program. As part of this initialization, the run-time support creates a file called the *shared memory file* for that program, and maps it into the process's address space where initialized and uninitialized shared data belong. The run-time then opens the parallel program's object file and copies the initialized shared data from the program's object file into the process's shared memory. At this point, this mapped memory contains that programs shared data and shared bss, which will be shared among all the processes forked from this parent process.

## 5.2. Shared Memory Expansion

Frequently, programs cannot predict their memory requirements at compile time. For example, a compiler cannot predict how big a symbol table it will need to compile a program. Such programs need to be able to allocate more memory at run-time. Standard UNIX provides `brk()` and `sbrk()` functions to grow or shrink a process's data segment. Library functions `malloc()` and `free()` provide heap management using memory allocated by `brk()` and `sbrk()`.

Parallel programs also need to be able to extend the size of their shared memory region dynamically. The interfaces used to manipulate shared memory in a parallel program are analogous to the standard UNIX interfaces to manipulate private memory. `Shbrk()` and `shsbrk()` functions grow the shared memory region. A process requiring more shared memory can call `shsbrk()`, specifying the number of additional bytes of shared memory needed. `Shsbrk()` maps more of that program's shared memory file into that process's address space. An `mmap()` system call is used to extend the size of that file if needed (see section 6.1).

The `mmap()` system call changes the address space of only the process that called it. When a process calls `shbrk()`, only that process has the newly allocated shared memory mapped into its address space. It is possible for a process to incur a SIGSEGV signal by referencing shared memory that one of its sharing partners has created, but hasn't yet been locally mapped. To solve this problem, the run-time library gives each process a SIGSEGV signal handler to map the additional shared memory into that process.

`Shmalloc()` and `shmfree()` library routines are available for managing shared-memory heap storage. If two processes call `shmalloc()` concurrently, they are guaranteed to get unique blocks of shared memory.

Figure 3 shows that the shared data is located above the private data in a process's address space. a gap in that address space is left between these two regions to allow for dynamic growth of private data. Any `brk()` system call which tries to extend the private data beyond the beginning of shared data will fail. The loader can be used to increase the size of this gap if an application needs more dynamic private memory growth (see section 4.4).

## 6. Operating System Support

Whenever possible, support for parallel programs is provided in user-level libraries. There are some support needs that must be provided by the operating system, however. This section discusses the basic operating system support that is used in building the parallel program run-time system, and kernel facilities to enable the construction of a debugger capable of dealing with parallel programs.

### 6.1. Mapped File Shared Memory

A flexible mechanism for providing shared memory in UNIX processes is necessary to support parallel programs. Several potential shared-memory interfaces were considered to support this need:

- The System V `shmop` interfaces [SVID 1986].
- The `mmap` interfaces mentioned in the original 4.2bsd System Manual [Joy 1982].
- A new set of interfaces chosen explicitly to support parallel programs.

The System V interfaces seemed awkward to use in the context of the parallel run-time implementation. They create a new object type with a name space disjoint from the file system name space, and an access protection mechanism different than that of the file system. In addition, the System V shared memory interfaces fail to allow extensibility of shared memory regions, which is very important in the implementation of the parallel run-time system.

Defining and implementing a unique set of interfaces may solve the immediate problem but has little chance of being compatible with any standard.

The 4.2bsd `mmap()` interfaces provide a means to map portions of a file into the virtual address space of a process. Given a file descriptor, `mmap()` can map a contiguous subset of the referenced file into continuous virtual addresses in the calling process. The file becomes a paging store for the virtual

addresses. If more than one process maps the same portion of a file into its address space, the processes are effectively sharing memory. This mechanism is cleanly integrated with UNIX since a file descriptor is used to specify the object to be mapped; the mapped objects appear in the file system name space, and no new object type is created. Further, depending on the type of file referenced by the file descriptor, `mmap` can create non-paged maps of IO devices, or special memories (e.g., bit-mapped graphics buffers). Additionally, it is possible to create the System V shared memory interfaces entirely in user-level libraries on top of the `mmap` mechanism; details are beyond the scope of this paper.

Sequent chose to implement a version of the 4.2bsd `mmap` interfaces, for reasons given above. The currently supported functions are:

**`mmap(addr, len, prot, share, fd, pos)`**

maps a subset of a file into a contiguous set of virtual addresses, extending the file if necessary to be at least as large as the map request. Previous content of the virtual address space, if any, is lost. Optionally, `mmap` can paint the address space with private zero-fill pages (i.e., non-shared address space that is initialized to zeroes on first reference).

**`munmap(addr, len)`**

unmaps a portion of process address space, replacing it with invalid addresses. A reference to address space that has been unmapped results in a segmentation violation signal (SIGSEGV).

Calls to `mmap` and `munmap` can be freely intermixed, resulting in a flexible mechanism to manipulate process address spaces. Mapping multiple contiguous portions of a file into similarly contiguous virtual addresses is treated essentially as a single `mmap` request. Thus if an application determines dynamically that more of a file needs to be mapped, it can extend the portion of the file mapped without consuming additional system resources.

The parallel run-time uses `mmap()` and `munmap()` system calls extensively to manage both private and shared memory in parallel programs. Detailed interfaces, semantics, and other applications of `mmap()` are beyond the scope of this paper.

## 6.2. Parallel Ptrace

A very real problem with programs is that they rarely run perfectly on their first try. This problem is worse with parallel programs, since in addition to functional or algorithmic bugs, they can suffer from bugs in synchronization, race conditions, and unrepeatable behavior. UNIX provides the `ptrace` (process trace) function to aid the debugging of single process applications. A number of debuggers have been created using the `ptrace` facility as the underlying mechanism to examine and manipulate a process (e.g., `adb`, `dbx`). However, as specified in standard UNIX, the `ptrace` facility is only useful between a process and its immediate child. If a traced child process forks, the new child is not visible to, nor controllable by, the original process. In addition, the only events in the child process visible to the parent are the signals it takes. This is useful for single-stepping and setting breakpoints, but leaves other interesting events such as forks unseen by the parent. Thus the standard `ptrace` mechanism is incapable of debugging truly parallel applications which create some number of processes dynamically, all working together in shared memory.

This problem was addressed by extending the `ptrace` function to make all signal, fork, exec, and exit events in a debugged child or any of its descendants visible to the debugger. Any descendant of a parallel process debugger may be stopped, examined, modified, and restarted. In effect, a parallel debugger has complete control over the hierarchy of processes running under it. Several new `ptrace` sub-functions are provided to support these functions. The new features are upwardly compatible with the original `ptrace` features; a debugger that doesn't use the new features sees exactly the old `ptrace` interface and semantics. These new functions enable the creation of a *parallel* version of `dbx`, `Pdbx`, which gives the programmer control over all threads of execution in a parallel program. The exact specification of the new `ptrace` mechanism and the parallel `dbx` debugger is beyond the scope of

this paper. More information is available in [PDBX 1986].

## 7. Higher Level Constructs

The following sections outline higher level constructs for developing parallel programs. These constructs are built using the facilities outlined above. Support for synchronizing processes, mutual exclusion, and a specific form of parallel programming called "microtasking" will be discussed below. A detailed example using these facilities is presented later.

### 7.1. Mutual Exclusion

Processes using shared memory to communicate need a mechanism to enforce mutual exclusion on accesses to shared variables. The parallel run-time system includes *s\_lock* and *s\_unlock* functions, which use hardware support to implement busy-wait P and V semaphores [Dijkstra 1968].

### 7.2. Process Synchronization

Sometimes, an application must ensure that all parallel processes have finished execution of a block of code before any process proceeds to the next block. To do this, the application can insert a call to a merge function at the end of the first block. This merge function is called a *barrier*. Processes arriving at and executing that barrier will execute a busy-wait loop until all cooperating processes arrive at that barrier. Then, all processes are allowed to proceed.

### 7.3. Microtasking Functions

The parallel run-time system provides an easy-to-use interface for applying a fixed number of processes to execute parallel code. Typically, each process executes the same body of program code as all the other processes, but applies that code to independent blocks of data. This model works particularly well for parallelizing program loops. For example, one iteration of a program loop which implements matrix multiplication multiplies the elements of a row in one matrix by the elements of a column in another matrix. The results of one loop iteration is independent of the results of previous iterations. Thus, several iterations of this loop can execute in parallel, with little need for communication among the processes. The processes typically synchronize to assign loop iterations to processes. These processes can synchronize using the lock and barrier functions outlined above.

The implementation of this model in the run-time system requires the developer to extract the parallel code from a program and put it into a procedure. The run-time system function *m\_fork()* is used to create a set of processes and start them executing that procedure in parallel. When the parallel function is complete, the parent process returns from *m\_fork()*. The other processes execute a busy-wait loop waiting for more work to do. Before and after calls to *m\_fork()*, the parent can execute sequential parts of the program, such as initialization and program cleanup. If there is more parallel work to do, the parent process can use *m\_fork()* again, to invoke another parallel function. *M\_fork* will reuse the processes created on the first *m\_fork()* call to execute succeeding parallel functions. This reduces the cost of creating processes.

While the child processes are busy-waiting for more work to do, they are consuming processor cycles. During the sequential parts of the program, the parent process can terminate or suspend the child processes by calling *m\_kill\_procs()* or *m\_park\_procs()* respectively.

### 7.4. Ada

The Ada language includes constructs to express parallelism. Ada *tasks* define parts of a program that can execute in parallel. Each task represents a thread of execution in the Ada program, complete with its own procedure-call stack, register context, and program counter. Logically, each Ada task can execute on a separate processor. There is no language-defined limit to the number of tasks an Ada program can create and have active. Ada *accept* and *call* constructs are the basis for synchronization and communication between tasks. Two tasks that have synchronized with each other are said to be in *rendezvous*. While in rendezvous, two tasks can exchange information.

An Ada compiler and run-time system have been created using the parallel programming run-time system which supports truly parallel execution of Ada tasks [Olien 1986]. Each of several DYNIX processes can choose a task to execute from a queue of runnable tasks which is maintained in shared memory. The Ada compiler was modified to place all program data structures into shared memory. Dynamic allocation of new Ada structures and task procedure-call stacks use the `shmalloc()` and `shfree()` functions. No Ada data structures are ever placed into process-private memory. Thus, a task can access its variables or other task's variables regardless which process is executing that task at the time. This is necessary to allow any task to be run by any process. In addition, it makes communication between tasks easy to implement. Synchronization between tasks is implemented using the process lock and barrier functions outlined below.

## 8. Example and Discussion

The example C program below uses most of the parallel programming facilities outlined above. This program implements a simple, naive algorithm for counting the number of primes between 1 and a maximum number. Determining if a number is prime involves substantial computation. This algorithm tests in parallel whether each of several numbers is prime, thus utilizing the computation power of several processors. In the example, there are four data structures which reside in shared memory. The shared variable `index` keeps track of the next number to check. The shared lock variable `index_lock` controls access to `index`. Variables `flags` and `counters` point to data structures allocated from shared memory at run time. `main()` is executed serially by the parent process. After completing some initialization, `main()` invokes the parallel procedure `work()` via the library routine `m_fork()`.

In the `work()` procedure each process selects a *bite* size range of numbers to check for being prime. Access to the variable `index` is controlled by asserting the lock variable `index_lock` before referencing `index`. The locking protocol imposes some run time cost on accessing `index`. Using larger *bite* values reduces the frequency of reference to `index`, and hence reduces this cost. Using too large a *bite* value relative to the number of values being checked could result uneven distribution of work among the processes. So, *bite* should be some small fraction of the maximum number being checked.

The sequential version of this program finds the primes between numbers 1 and 100000 in 865.1 seconds elapsed time. Using 28 processors and a *bite* size of 512, the same range of numbers is checked in 34.5 seconds. This is a speed up of 25X using 28 processors.

```

main(argc,argv)
int argc;
char *argv[];
{
    register int nprocess; /* number of processes */
    register int maxprime; /* maximum number to test */
    register int i;
    register int nprimes; /* number of primes found */
    int *counters; /* counter for each process */
    int bite; /* numbers taken at a bite */

    if (argc != 4) {
        printf("Usage: prime bite nprocess maxprime/n");
        exit(1);
    }
    bite = atoi(argv[2]);
    nprocess = atoi(argv[3]);
    maxprime = atoi(argv[4]);

    /* allocate counter array in shared memory */
}

```

## Parallel Programming Example

```

#define TRUE 1
#define FALSE 0

/*
 * SYNOPSIS
 *   prime bite numprocs maxprime
 * DESCRIPTION
 *   Finds the prime numbers between 1 and 'maxprime',
 *   inclusive, using a parallel algorithm to
 *   find them. 'numprocs' specifies the number of
 *   processes working in parallel to solve the problem.
 *   'bite' specifies how many numbers are bitten off at
 *   a time by a process to check for prime.
 */
#include <stdio.h>
#include <errno.h>
#include <math.h>
#include <parallel/parallel.h>

shared int      sindex;          /* shared, next number */
shared slock_t  sindex_lock;     /* program-wide semaphore */
extern int      m_myid;          /* index number of a process */
/* (parent process is index 0 */
work();          /* the parallel function */

/*
 * main() is executed serially by the parent process.
 */
main(argc,argv)
    int      argc;
    char     *argv[];
{
    register int numprocs; /* number of processes */
    register int maxprime; /* maximum number to test */
    register int i;
    register int numprimes; /* number of primes found */
    int         *counters; /* counter for each process */
    int         bite;      /* numbers taken at a bite */

    if (argc != 4) {
        printf("Usage: prime bite numprocs maxprime\n");
        exit(1);
    }
    bite = atoi(++argv);
    numprocs = atoi(++argv);
    maxprime = atoi(++argv);

    /*
     * allocate counter array in shared memory
     */

```

```

counters = (int *)shsbrk(numprocs * sizeof(int));
sindex = maxprime;
s_init_lock(&sindex_lock);

/*
 * creat (numprocs - 1) processes and send parent
 * and child processes into work().
 */
m_set_procs(numprocs);
m_fork(work, bite, counters);
m_kill_procs();
/*
 * Sum number of primes found by each process
 */
numprimes = 0;
for (i = 0; i < numprocs; i++)
    numprimes += counters[i];
printf("number of primes is %d\n", numprimes);
exit(0);
}

/*
 * WORK:
 * All processes enter here to work on the problem.
 *
 * While bites remain:
 *     Lock semaphore.
 *     Allocate a range of numbers to check by decrementing
 *     shared index, copy global index into local
 *     index.
 *     Unlock semaphore.
 *     Then check each index in this bite for prime.
 *
 */
work(bite, counters)
register int bite;
register int *counters;
{
    register int i;
    register int lindex; /* local index for flags */

    /*
     * Grab "bites" and check for primes.
     */

    lindex = 1;
    while (lindex > 0) {
        s_lock(&sindex_lock);
        sindex -= bite;
        lindex = sindex;
        s_unlock(&sindex_lock);
    }
}

```

```

        for (i = lindex + bite;
             (i > lindex) && (i > 0); i--) {
            if (is_prime(i) == TRUE)
                counters[m_myid]++;
        }
    }

is_prime(number)
    int    number;
{
    register    i, sq;

    if ((number & 1) == 0)
        if (number == 2)
            return(TRUE);
        else
            return(FALSE);

    sq = (int)(sqrt((double) number) + 1);
    for (i=3; i<sq; i+=2)
        if ((number % i) == 0)
            return(FALSE);

    return (TRUE);
}

```

## 9. Performance

The facilities outlined above help developers construct parallel applications. One of the main reasons for constructing parallel applications is to achieve significantly better performance than can be achieved with only one processor. A number of factors influence the performance of a parallel application. A few of these factors are outlined below.

### 9.1. Amdahl's Equation

Modifying a program to use parallelism involves determining which parts of the program can run in parallel, and which must run sequentially. For example, initialization, I/O operations, and program completion frequently must be performed sequentially. If this sequential code represents too large a fraction of the sequential program's total run time, it will dominate the program's run time after it has been parallelized. UNIX profiling can be used to determine how much time is spent in various parts of a sequential program. Amdahl's equation can be used to calculate the maximum theoretical speedup obtainable from parallelizing a sequential program. For some program, let

- T be the elapsed time the sequential version of that program takes to execute.
- N be the number of processes executing the parallel version of the program.
- s the fraction of sequential execution time T executing code that must remain sequential.

The sequential part of the program will execute in time

$$s * T$$

Minimum time for execution of the parallel part of the program will be

$$(1-s)*T/N$$

For example, suppose the sequential version of a program runs in 100 minutes, 5 percent of its code must remain sequential, and ten processes are applied to running its parallel version. The parallel version would run in about 15 minutes. This represents about a 7X speedup. Fortunately, for many programs, the code that must run sequentially is less than 1 percent. Using 1 percent sequential execution time in the above example, parallel execution time drops to about 11 minutes, producing a 9X speedup. With larger numbers of processors applied, the percent sequential code has increasingly larger effects on the maximum speedup. Applying 20 processors to an application with 1 percent sequential code yields an execution time of 5.95 min, a maximum speedup of 16.8X. Notice that with an infinite number of processors applied, the minimum execution time for a parallel program will be  $s \cdot T$ , for a speedup of  $1/s$ . Thus, limit to the speedup in a program with 1 percent sequential code is 100X.

## 9.2. Synchronization Costs

Adding locks and barriers to an algorithm for mutual exclusion and synchronization adds to the execution time of a parallel program. The lock and barrier primitives take some small amount of time to execute. When two or more processes attempt to assert the same lock variable simultaneously, those processes are said to have *collided* on that lock variable. When two or more processes collide on a lock variable, one process is granted the lock. The other processes execute busy-wait loops until it is their turn to be granted the lock. The processing time wasted due to collisions is influenced by the frequency of collisions, and by the length of time each process holds the lock once it is granted. For these reasons, parallel programmers should take care that locks are used only when necessary, and that they are held for as short a time as possible.

Two techniques to accomplish this goal can be seen in the primes program above. Notice that each process works on a *bite* size range of numbers for prime values between references to *sindex*. This reduces the frequency of assertions of *sindex\_lock*. The prime example program using 28 processors and a *bite* size of 1 searches from 1 to 1000000 in 40.8 seconds. This program takes only 34.5 second, using a *bite* size of 512. Notice also that *sindex\_lock* is held only long enough to decrement *sindex*.

Also notice that each entry in the shared memory array pointed at by *counters* is used by only one process as a counter for the number of primes that process finds. When the parallel work is finished, the parent task then sums the counters for each process to determine the total number of primes. This technique avoids the need for locking and unlocking a global counter each time a prime is found.

## 9.3. Balanced Distribution of Work

Another requirement for good parallel performance is that work be evenly distributed among the processes. In the prime example, one might set *bite* equal to *maxprime* divided by *numprocs*. With only one reference to *sindex*, each process would search the the same size range of numbers for prime numbers. But, small numbers can be tested for being prime much more quickly than larger numbers. The processes testing smaller numbers would finish long before those testing larger numbers, and would busy-wait for the later ones to finish. Much potential parallelism would be lost. In the prime example program, using 28 processors and a *bite* size of 25714 to search numbers from 1 to 1000000 runs in 44 seconds, vs 34.5 seconds for a *bite* size of 512.

## 10. Problems and Futures

The above model for developing parallel programs works for a wide variety of applications. There are a number of areas where improvements could be made. Below is a brief discussion of these areas.

### 10.1. Co-scheduling

Parallel programs running on DYNIX use standard DYNIX processes to execute some algorithm in parallel. The performance of a parallel algorithm can suffer if these processes are not actually executing concurrently. For example, suppose a process is holding a lock and is preempted from its processor by the operating system. Other processes trying to assert that lock will busy-wait until that process is scheduled to run again and releases the lock. Lots of processor time could be wasted in this manner. DYNIX provides simple mechanisms for encouraging these processes to execute concurrently (such as turning off priority aging, etc. [Sequent2 1986]), i.e. to co-schedule those processes. These mechanisms work well when there is only one parallel application which is expected to run continuously until completion. Sharing a computer among several parallel applications, or between a heavy time-sharing work load and a parallel application can yield poor performance.

Improvements in the support for time-sharing parallel applications might be useful. Ideas include:

- use a daemon process to force parallel applications to efficiently time-slice their use of the system. The daemon could ensure that there are enough computer resources available to run each parallel application efficiently before allowing that application to run.
- integrate the notion of a parallel application into the DYNIX processor scheduling algorithm. The parent process could identify itself as the root of a parallel application. Processes forked by that parent could then be co-scheduled by DYNIX.

### 10.2. Buffered I/O

The UNIX standard I/O library provides a set of I/O subroutines, such as `fread()` and `fwrite()`, which buffer I/O data transfers. Buffering reduces the number of read and write system calls performed by that process. Larger chunks of data are passed to the kernel with each system call. This reduces operating system overhead for I/O operations.

The standard I/O library allocates memory for I/O buffering in a process's private memory. In a parallel application, each process has its own private standard I/O buffers. If two processes are writing to the same file, this private buffering may cause the order of data written to the file to be significantly different from the order in which the application produced the data.

A possible solution would be to provide a standard I/O library which buffers I/O in shared memory. Locks would be required to synchronize access to the buffer.

### 10.3. Shared File Descriptors

Each UNIX process maintains a table of files that process has open. The *file descriptor* argument to system calls such as `read()` and `write()` is merely an integer index into that process's open file table. Each open file has associated with it a current offset. This identifies the next byte in the file to be read or written.

In the current implementation, each process running a parallel program has its own view of the set of files that process has open. There is no coordination of these views across all the parallel application. If one process opens a file and passes the file descriptor to another process for doing I/O, unpredictable results will occur.

The parallel Ada implementation discussed above uses a set of library routines to emulate *shared file descriptor* I/O. This emulation uses a *global file table* located in shared memory which contains what the program-wide view of each file should be. This table has its own lock for mutual exclusion. The table contains information about the file, such as the name of the file, what the current "program wide" offset for that file is, etc. Processes doing I/O use this table to ensure that process's private view of the file is up to date before doing any I/O to that file. After performing its I/O, that process updates the table to reflect the new status of the file. This I/O emulation is somewhat awkward and indicates a need for a better solution, possibly integrated with a process co-scheduling solution.

#### 10.4. Parallel Program Initialization

In the current implementation, the run-time library allocates and initializes the shared memory regions of the process. This involves opening and reading parts the program object file into shared memory. The run-time initialization code uses PATH environment variable and argv[0] to construct a file name suitable for opening the program object file. Under some circumstances, this construction technique can be confused. A more robust and efficient implementation of this function seems desirable. One possible solution would be for the DYNIX kernel's exec system call to do this initialization.

#### 11. Summary

This paper presented a process model for executing parallel programs on a shared memory multiprocessor. The model involves a simple extension of the standard UNIX process model, a set of extensions to languages and run-time system support, and some operating system support. The resulting system is capable of supporting a variety of higher level parallel programming constructs in several languages (e.g., microtasking and Ada tasking), and can yield good parallel program performance. It allows programmers to concentrate on parallel algorithms instead of details of creating parallel processes and shared memory.

## References

- [Beck 1985]  
Beck, Bob, and Bob Kasten, "VLSI Assist in Building a Multiprocessor UNIX System", Conference Proceedings, Summer Usenix 1985.
- [CSTD 1985]  
Preliminary Draft ANSI Standard for the C Language, X3J11/85-102, November 11, 1985.
- [DOD 1983]  
United States Department of Defense, "Reference Manual for the Ada Programming Language", ANSI/MIL-STD-1815A-1983, February 17, 1983.
- [Fielland 1984]  
Fielland, Gary, and Dave Rodgers, "32-Bit Computer System Shares Load Equally Among Up to 12 Processors", Electronic Design, September 6, 1984.
- [Gehani 1985]  
Gehani, A. H., and W. D. Roome, "Concurrent C -- An Overview", Conference Proceedings, Winter Usenix 1985.
- [Joy 1982]  
Joy, William, et.al., "4.2BSD System Manual, Draft of September 1, 1982", Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley.
- [Nicolau 1981]  
Nicolau, Alexandru, and Joseph A. Fisher, "Using an Oracle to Measure Parallelism in Single Instruction Stream Programs", 14th Annual Microprogramming Workshop, ACM Special Interest Group on Microprogramming, October 1981.
- [PDBX 1986]  
Sequent Computer Systems, "DYNIX PDBX Debugger User's Manual", May 2, 1986.
- [Rodgers 1985]  
Rodgers, David P., Ian L. Johnstone, Sandra L. Baldrige, "Multiprocessing in a Software Development Environment: Parallel Tools", Wescon 1985 Proceedings.
- [Seitz 1984]  
Seitz, Charles L., "The Cosmic Cube", Communications of the ACM, Vol. 28 No. 1, January 1985.
- [Sequent 1986]  
Sequent Computer Systems Technical Note, "Parallel LINPACK Case Study", January 20, 1986.
- [Sequent2 1986]  
Sequent Computer System, "Guide to Parallel Programming", Portland, OR. Sequent Computer Systems, Inc., 1986.
- [SVID 1986]  
System V Interface Definition, Issue 2, Volume 1, AT&T, 1986.
- [Olien 1986]  
"Parallel Ada Tasking on the balance 8000", Winter 1987 Uniforum Proceedings
- [Dijkstra 1968]  
"Cooperating Sequential Processes", Programming Languages, pages 42-112. Academic Press, New York, New York, 1968

# **A Prototype Capacity Planning and Configuration Modeling Tool for UNIX® Systems**

G. Ramamurthy  
Y. T. Wang  
AT&T Bell Laboratories  
Holmdel, New Jersey

Hank Nichols  
Mike Andrews  
AT&T Information Systems  
Basking Ridge, New Jersey

## **ABSTRACT**

Modern computer systems are rapidly evolving into complex systems, and there is a constant need for tools and techniques to understand and predict the performance of these complex systems. This paper presents a prototype capacity planning and configuration modeling tool for the AT&T 3B series of computers running under UNIX System V.

## **1. INTRODUCTION**

This paper presents a prototype capacity planning and configuration modeling tool for the AT&T 3B<sup>1</sup> series of computers running under UNIX<sup>2</sup> System V, and the application software supported by AT&T. The tool is implemented on the AT&T PC 6300 and can be used by AT&T marketing/sales personnel and value added retailers of AT&T products. It determines a hardware configuration that is capable of handling the projected workload with acceptable response time performance. It can also be used to study the effects of workload and/or hardware changes on the performance of a given system. With the menu driven interface, the tool can be used by users interactively with minimum expertise.

Modern computer systems are complex and rapidly evolving, and there is an ever increasing need for tools and techniques that will assist in understanding the performance of these complex systems. Capacity planning is the process of understanding and predicting the performance of computer systems running a variety of applications. Such an understanding is necessary to provide intelligent answers to the questions of cost and performance that arise throughout the life of a system. For existing installations, capacity planning and prediction essentially determine if the computers have sufficient processing capacity to handle not only the current workload, but also determine if enough reserve capacity is available to handle projected workload growth. It is necessary to know where the system bottleneck would be, and the relative cost effectiveness of various alternatives for alleviating performance problems. For a new installation the task consists of determining a hardware configuration that will handle the projected or estimated workload

---

1. 3B is a trade mark of AT&T

2. UNIX is a registered trade mark of AT&T

with acceptable levels of performance and cost. These issues are of great significance to the customer and serious repercussions can result from poor estimates and recommendations.

## **2. METHODOLOGY**

To settle the above issue, one must have a good understanding of the system under consideration, the applications supported by the system, and the expected use and performance of the system. One solution is to determine the alternatives through experimentation. Experiments however are expensive and laborious to perform. Further, although experiments may yield accurate results with assumed workload, they may fail to offer insights into the system. For existing installations, measurement data collected from the system can provide a great deal of information but require quite an expertise in interpreting them. Neither of the above solutions can predict performance when one or more components of the system are changed. Our solution to the problem is to combine the advantages and strengths of benchmarking/measurement and the state-of-the-art modeling techniques. In particular, we use queueing network models that have been proven to be an effective tool in the design and analysis of computer systems. To obtain workload parameters for use by the model, we use benchmarks that are representative of the way a typical user would use the application.

## **3. THE CAPACITY PLANNING AND CONFIGURATION MODELING TOOL**

Our capacity planning and configuration tool has been developed with the following methodology and is intended to help AT&T marketing personnel make proper choice of new configuration, and evaluating the performance of existing installations. A key ingredient in the capacity planning process is workload characterization. For an existing installation, the workload can be characterized from measurement data. The existing UNIX System V measurement packages like the System Activity Report, Accountcom, etc are used. The measurement parameters are then mapped into parameters for a performance prediction model. To aid the process of measurement and subsequent conversion of measurement data into model parameters, a Performance Engineering & Management tool has been developed [1].

In the case of a new installation for which no measurement data is available, workload characterization is a complicated process. To overcome this problem, for selected application software supported by AT&T, benchmark scripts that are representative of the way a typical user would use the application are first developed. With the help of such benchmark scripts, workloads for each supported application software are characterized with a proprietary remote terminal emulator. The resulting workload data is then converted into a form useful to the Capacity Planning and Configuration Modeling Tool running on the AT&T PC6300, using the Performance Engineering & Management tool. The characterization is also performed at several different dimensions including, light, medium and heavy usage of a given application. Commands or functions in each application are grouped into several unique classes like simple commands, complex commands, etc. Such a classification can be based on the response time for a given command, its actual function or

operation, the type and amount of resource usage, etc. For a new customer installation, a set of appropriate workloads are then chosen to approximate the way the customer intends to use these applications.

At the heart of the tool is a performance prediction model of the UNIX operating system. The model is a non-product form priority queueing network model that is solved analytically [2]. The model can handle multiple applications running concurrently. Given a configuration, the associated device parameters, the set of workloads (or applications) that has to be supported concurrently, the number of users associated with each application, terminal transmission rates and user typing rates, the model predicts the mean response time for each class of command in each application, the mean throughput of each application, and the device utilizations for each application. The model can also be used to estimate the effects of workload and/or hardware changes on performance, and thus configure a system with adequate performance and an acceptable cost per user for the customers.

The prototype tool is a user friendly, interactive tool with extensive menu driven capability. The tool prompts the user for input and does extensive checking with respect to compatibility, constraints on marketable configurations, configuration constraints like disk storage, main memory, etc. Figure 1, shows a sample screen input when the chosen application is a spread sheet (20/20)<sup>3</sup>. The figure shows that there are four users in this application, all of them working on small files, the user typing rate is 30 words per minute and the terminal transmission rate is 4800 bauds. The figure also shows that the four users together would need approximately 930 kilo bytes of main memory. Figure 2 shows a sample screen summary output, where, three applications (Crystal Writer Plus,<sup>4</sup> Informix,<sup>5</sup> and 20/20) are assumed to be running concurrently on a system with two hard disks and four mega bytes of main memory. The screen displays the number of users in each application, the file size used by each application, the user typing rates, the user terminal transmission rates and the estimated main memory requirement of each application. At the bottom of the screen the total main memory required (including the UNIX kernel) to support all three applications and the indicated number of users in each application concurrently is displayed graphically. Once the required data has been entered and verified, response time and system utilization results for the chosen configuration and application/user mix are generated within seconds (see figure 3), such that the tool can be used on an interactive basis. Many enhancements have been planned for the tool in future versions. We believe this tool will help both the vendor and the customers to better engineer and manage the UNIX based system performance.

---

3. 20/20 is a trade mark of Access Technologies

4. Crystal Writer Plus is a trademark of Syntactics

5. Informix is a trade mark of Informix Systems

## REFERENCE

1. B. L. Farrell and G. Ramamurthy, " A Prototype Performance Engineering/ Management Tool for UNIX Based Systems", International Conference on Management and Performance Evaluation of Computer Systems, Las Vegas, December 1986.
2. G. Ramamurthy, " An Analytical Model of a Non-reentrant Operating System", submitted for publication for the 1987 ACM SIGMETRICS Conference.

187.0	20/20
278.0	How Many Concurrent Spread Sheet Users? 4
089.0	S = 25 ROWS/ 40 COL ( 16K )
530.2	M = 100 ROWS/ 40 COL ( 64K )
	L = 1000 ROWS/ 40 COL ( 250K )
	ENTER < S, M or L > for FILE SIZE: s
	ENTER TYPING SPEED ( 1 THRU 160 ) IN WORDS PER MINUTE: 30
	ENTER LINE SPEED ( 300, 1200, 2400, 4800 or 9600 ) BITS PER SECOND: 4800
	20/20 users require 0.9300 Meg of MAIN MEMORY
	THIS DOES NOT INCLUDE USER DATA
	Press the ANY KEY to continue
MAIN MEMORY	
DISK SWAP AREA	
....1...2...3...4...5	
!!!!\$\$\$\$\$\$	MEG

Fig. 1: Sample screen input for a spread sheet application.

APPLICATION	USERS	RESPONSE TIME	UTILIZATION CPU	DISK	MEMORY
Crystal Writer Plus	4	-	-	-	0.731
INFORMIX	3	-	-	-	0.372
20/20	4	-	-	-	0.930
TOTAL	11				2.033

Press the ANY KEY to continue

MAIN MEMORY

DISK SWAP AREA

....1....2....3....4....5  
!!!!!!\$\$\$\$\$\$\$\$

Fig. 2: Sample screen summary output with three applications.

APPLICATION	USERS	FILE SIZE	TYPING SPEED	LINE SPEED	MEM	DISK
Crystal Writer Plus	4	20 PAGE DOCUMENT	30	9600	0.731	2
INFORMIX	3	10,000 RECORDS/ 16 FIELDS	30	9600	0.372	2
20/20	4	25 ROWS/ 40 COL	10	4800	0.930	1
TOTAL	11				2.033	MEG

Press the ANY KEY to continue

#### MAIN MEMORY

#### DISK SWAP AREA

.....1...2...3...4...5  
 ##### MEG

Fig. 3: Sample screen display of results with three applications.

# A Knowledge-based System for Performance Tuning of the UNIX<sup>®</sup> Operating System

*Behrokh Samadi  
AT&T Bell Laboratories  
Crawfords Corner Road  
Holmdel NJ 07733*

## ABSTRACT

Tuning an operating system for better performance is an ongoing task of the system administrator. It includes activities such as adjusting operating system tuning parameters, running maintenance routines, developing operation rules, and modifying hardware. However, this problem is extremely workload dependent and the documented rules tend to be general. Successful tuning requires interpretation of large volumes of data, problem detection using deep knowledge of operations and internals, and making changes that result in reasonably predictable outcomes. As a result of a shortage of expertise and time, systems may operate with minimal tuning. Therefore, a tool that helps in simplifying this process could be of great benefit.

Herein we describe Tuning Aid, an automated tool to tune the UNIX<sup>1</sup> operating system. The Tuning Aid includes tuning rules and a number of performance modules to diagnose performance problems and make specific recommendations and predict the resulting change in performance.

### 1. Introduction

This paper describes a research project currently underway to evaluate the potential of using an automated tool to tune the UNIX operating system. Tuning a computer system, which is often aimed at improving its performance, involves several activities. The tuning activities we are referring to in this paper include, a) adjusting some operating system parameters, such as number of buffers; b) running maintenance routines, such as *dcopy*; c) developing operation rules, such as off peak hour runs of backups; and d) modifying hardware, such as buying an additional disk drive.

An operating system tuning task consists of following subtasks. First, the measurement data is to be interpreted to detect areas with performance problems or improvement possibilities. Due to the large amount of data, this is usually a time consuming task. Next, if a problem is detected, an appropriate tuning action is to be taken. To detect problems and find a remedy, a fair amount of system's operations and internals knowledge is needed. Finally, the outcome of a tuning action

---

1. UNIX is a registered trademark of AT&T.

is either measured by monitoring the system after the change, or, it is predicted using a performance model. The former requires time and resources and the latter requires performance modeling and analysis knowledge. In any case, if the tuning action is unsatisfactory, it needs to be revised.

There are a number of tuning rules which help users in the tuning process. However, we find that while certain conditions can be detected with the use of simple rules, in order to obtain an adequately tuned system, a substantial amount of quantitative knowledge needs to be codified into somewhat sophisticated models. If possible, accompanying a recommendation, should be a prediction of the quantitative extent of improvement. For example, a response time reduction of 70% and 5% are very different results if they require expenditure on additional equipment or time consuming operations. In summary, for successful tuning the necessary knowledge consists of data analysis, system operations and internals, and modeling and performance analysis. However, a shortage of expertise in these areas and the complexity of system administration functions often result in badly tuned systems. The problems suggest incorporating these areas of expertise into a knowledge-based system for performance tuning.

In this paper, a knowledge-based system, called the Tuning Aid, is proposed. The Tuning Aid includes sufficient knowledge of UNIX internals, data analysis and performance modeling to a) diagnose performance problems or detect areas for improvements, b) make recommendations for tuning, and c) learn from previous successes and failures in order to make new recommendations. In the paper we also present some details of the accomplished work in this regard. The Tuning Aid can be further integrated into a general administrative expert system which, in addition to the tuning expert, could have other subsystems, such as a networking expert, a security expert and an operations expert.

The idea of a computer system administration and performance management expert has been the subject of much recent work. One of the reported successful experiments, YES/MVS, helps operators of MVS installations in system operations [KAR 84]. Examples of other attempts in the area of computer performance related expert systems include: YSCOPE, a shell for building expert systems for computer systems performance [HEV 85]; an expert system for performance management of MVS installation [LEV 85]; and an expert system for computer systems capacity planning [SBD 85]. Given the importance and complexity of performance issues related to computer systems tuning, configuration and capacity planning, more research and development in these areas are expected to appear in the future.

In the next section, the structure of the Tuning Aid and the models which provide quantitative information are described. In section 3 the conclusions and an overview of future work in this area are presented.

## *2. Description of the Tuning Aid*

The UNIX operating system (System V) has several built-in utilities which collect and report the system measurements. The System Activity Report (sar(1)), the System Activity Disk Profile (sadb(1)), and the Accounting Commands (acctcom(1)) are examples of some of these utilities. The data provided by these routines can be used to characterize the system's workload and study its performance.

The Tuning Aid is a rule-based system which receives the system measurement data, tuning history and information from several performance analysis modules to detect the problems and make quantitative recommendations (Figure 1). The tuning rules in the system consist of a) general tuning rules described in the tuning and administration manuals, such as:

if (the cpu is idle less than 10% of the time) then  
the cpu is *highly* utilized

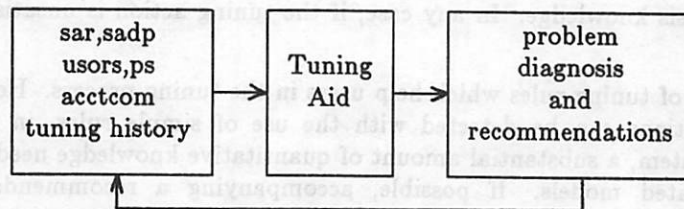


Figure 1. Tuning Aid.

b) the rules obtained by analyzing the performance of specific area of the operating system, such as:

if (a drive's utilization is more than 50% and  
any of the file systems on that drive is *unorganized*) then  
recommend performing a dcopy on the unorganized file systems;

and c) the rules obtained by analysis of the measurement data, such as:

if (a problem is detected and the cpu is *highly* utilized) then  
make recommendations;  
if (a problem is detected and the cpu is not *highly* utilized) then  
issue warnings;

The question of what is considered *high* utilization or what is the measure for *unorganized* file system is determined by different sources of knowledge. Some of these values are given as thresholds by the above general tuning rules. Others, such as a measure of disorganization of the file system, are derived by analyzing some performance models. It is expected that at a later stage, the system will be able to learn from its history and tune itself by adjusting these values.

The Tuning Aid can be invoked at any time the user desires to examine the state of the system. The measurement data during the requested period is analyzed and the observations, detected problem areas, and recommendations are produced. Accompanying each problem is a statement of the time period during which the problem was observed. The problem is considered vital if it appears during high resource utilization periods. In such cases, the Tuning Aid makes definite recommendations. However, a symptom may be observed during lower utilization periods. For example, an unorganized file system may tend to be used infrequently or during less busy periods. In such a case, the Tuning Aid only warns the user of the unorganized file system.

The overall structure of the Tuning Aid has been divided into multiple levels. The depth of a level specifies the extent of details studied to provide the information. Level 1 basically contains the very general tuning rules obtained from the tuning manuals and experienced system administrators. The quantitative knowledge in these rules typically appear in the form of thresholds (example above). In level 2, some analytic modules provide the system with more specific information and the ability to gauge the degree of improvement (especially whether or not significant) of any change. Finally, level 3 learns from the performance history and previous recommendations in order to adjust the threshold values, fill in missing information, and make better suggestions.

### 2.1 Level 1

The first step in the development of the Tuning Aid was to integrate into a package the general tuning rules described in the tuning and system administration manuals [UNI 84], [UNI 85]. At this level, the data from the System Activity Report is read and following the tuning rules, problem areas are detected and recommendations are made.

The level 1 process can detect some problem areas and make recommendations, although the recommendations are not specific enough and require further analysis. The followings are examples of some rules at this level:

if (the cpu load is high and the wait for IO is high and  
the cache hit ratio is low) then  
recommend increasing the number of buffers;

The data show that the cpu is spending too much time waiting for IO, a condition which *may* be improved by increasing the number of system buffers.

if (the cpu load is high and the wait for IO is high and  
the cache hit ratios are high and the disks are underutilized) then  
diagnose that tape or other IO operation may be in progress;

Here, the cpu seems to be slowed down by IO operations but the disk drives and buffer activity indicate few disk IO operations. The diagnosis is that tape operations is probably the cause of this slow down. This diagnosis is confirmed by searching the command accounting files.

The first level of the Tuning Aid has been developed and is running on several machines. This program, , reads the data during the requested measurement period, diagnoses problems and makes recommendations. The time period during which a problem was observed in the system is listed with every observation. Clearly, the severity of a problem depends on its frequency and duration of occurrence. It should make a difference if a problem existed 20 minutes or 2 hours. Nevertheless, a short-lived problem during critical operation periods may not be negligible. At the present time, the recommendations are made independent of the duration of time the problem persisted. The user should decide whether the recommended action needs to be taken. In the future, when the Tuning Aid learns from its history, it will be able to aid the user more in making this decision. Figure 2 gives a sample output from the first level program.

#### 2.1.1 Choice of Language

Initially the level 1 was written in C. It was then translated into OPS5 and further enhanced by June Hsiao [HSI 86] in an effort to build the system in an expert system development environment. The tradeoffs can be listed as follows:

- Knowledge representation and manipulation - Representing the tuning rules in the form of productions seems to be more suitable to our application. It is easier to code and modify an expert's knowledge in a non-procedural language.
- Run time efficiency - The running time of the C version is far less than the OPS5 version. Considering the fact that a tuning aid is most useful on heavily loaded machines, a more efficient language is a plus.
- Portability - The program needed to be tested on several machines. There are fewer testing sites that have both OPS5 and disk measurement utilities<sup>2</sup>.

Observations and recommendations between hours 8:00 and 18:00 on 7/11/86.

Observation (1): No problems were detected. The workload is cpu bound.  
Duration: 219 minutes during high cpu and IO activities and  
0 minutes during less busy periods.  
Recommendation: Reduce load.

Observation (2): The system is swapping out.  
Duration: 321 minutes during high cpu activities and  
0 minutes during less busy periods.  
Recommendation: Increase memory or reduce load.

Observation (3): 92% of the text table was used.  
Duration: 40 minutes during high cpu activities and  
0 minutes during less busy periods.  
Recommendation: Increase size of text table.

Observation (5): 90% of the file table was used.  
Duration: 120 minutes during high cpu activities and  
0 minutes during less busy periods.  
Recommendation: Increase size of file table.

Observation (6): The size of the hash table is large.  
Recommendation: It can be decreased to 256.

**Figure 2.** A Sample Output from the Level 1 Program.

- 
- Interface with other modules - A number of performance modules which provide information to Tuning Aid have been developed in C. These modules are computation intensive and require a number of UNIX system calls and library functions. The extension of the first level program to interface with these modules is easier if this program is also written in C.

Considering the efficiency and the portability of the C version, the C program is now running on most testing sites. At the same time, other UNIX based development tools are also being considered.

## 2.2 Level 2

The high level view of tuning in level 1 needs to be complemented with more specific and detailed information about the state of the system. The major difficulty is that few quantitative conclusions can be made by simply following general tuning rules. It is observed that while some conditions can be detected with these rules, substantial amount of information is hidden which can be obtained with more sophisticated methods.

- 
2. Disk drivers on some hardware do not provide the required disk measurement data such as those produced by `sadp(sadp(1))` and `sar` with `-d` option (`sar(1)`).

At level 2, quantitative analysis of some areas of the UNIX operating system are employed and coded into a number of modules. The analysis, depending on the subject, are predictive or empirical. Each module can be considered as a knowledge source which uses system modeling and analysis with operating system internals knowledge and provides more information to the Tuning Aid. The input to every module is obtained from the saved system measurement data or the real time data. Using these modules, we are able to add more rules to the rule base and make quantitative recommendation. For example, if a module analyzes the sensitivity of the size of the buffer pool to the workload, the following rule can be added to the rule-base:

if (cache hit ratios are low) then

analyze effects of increasing number of buffers;

The result of such an analysis is a set of pairs  $(x, y)$ , where  $x$  is the increase in number of buffers and  $y$  is the expected improvement in cache hit ratio.

The following sections cover a brief discussion of the modules that have been developed at this level. Each section accompanies the experimental result that were conducted to validate the modules. The details of each module appear in future documents.

### 2.2.1 Text Bit Module

This module provides a listing of the commands recommended for setting text bits (sticky bits).

Setting the text bit in a command often results in faster invocation of that command. On the UNIX operating system with swapping<sup>3</sup>, when a command (without the set text bit) is invoked, its text (object code) is brought into the main memory one block at a time. This requires multiple interrupts to the cpu and multiple requests to the driver. With a set text bit, the text after its first invocation, is stored in a contiguous space on the disk (swap space). On further invocations, the text is transferred into the main memory with one or a few read operations. The disadvantages are usage of contiguous disk space (reserved for the swapping operation), and an entry for every command with set text bit in some internal operating system table.

The commands that are expected to be reasonable candidates for setting text bits are those that a) are invoked frequently, but are not in the main memory most of the time, and b) are large and hence require many disk accesses if not copied from contiguous disk space.

Conventionally, a system administrator has to read the summary of command invocations and select the ones that seem to satisfy (a) and (b) above. To automate this procedure, a program is developed which reads the command invocation history and produces a listing of the commands in the order recommended for setting text bits. The order of the command in the list is determined by the function,

$$f(N, s, \rho()) = g(s) * \sum_{n=1}^N \frac{1}{1 - \rho(t_n)}$$

where

3. The treatment of texts with set text bits on UNIX with demand paging varies with different implementations, and its discussion is beyond the limits of this paper.

$N$  = number of times the command was invoked and a copy was not available in the main memory (from acctcom)  
 $g(s)$  = a function of  $s$ , the size of the command's text  
 $\rho(t)$  = measured cpu utilization during a period around  $t$  (from sar)  
 $t_n$  = time of the  $n$ -th invocation of the command when a copy was not in the main memory

Note that a third consideration is added to (a) and (b) above. That is, the commands that are invoked during busy periods should be given priority in the list.

This program needs to be run every day to provide a listing for that day. The run time of the program depends on the size of the acctcom files. The listings for different days are then merged and result in a single list once they converge. In case that the lists do not converge, a selection is made based on commands that appear on majority of the lists.

#### 2.2.1.1 Experimental Results

The text bit module was tested in a controlled environment on an AT&T 3B2/400, using a benchmark. This benchmark creates a number of terminal scripts as specified by the user. Each script runs several UNIX commands to resemble a session on the terminal. The benchmark driver collects some measurement data that is used in our experiments.

Identical benchmark runs were conducted under three configurations. First, the benchmark was run on a system with no commands with set text bits. The results of this configuration is shown as NTB in Table 1. The text bit program was then run using the accounting information from the testing period. The text bits of the first five commands suggested by the text bit program were then set. Second, same benchmark was run on the system with the set text bits. The results are shown as TB1 in Table 1. Finally, as the third configuration, the text bits on the commands were then changed to those commands recommended by the rules in the tuning manuals. The same benchmark was run after the last change. The results are shown in Table 1 as TB2.

TABLE 1. Benchmark Results with and without Text Bits

No. of Scripts	Elapsed Time	System Time	Wait for IO
1 NTB	265.30	75.98	67.12
TB1	253.62 (4.4%)	71.21	59.89
TB2	262.37(1.1%)	72.03	67.59
4 NTB	634.30	225.18	31.08
TB1	609.83 (3.9%)	211.91	19.82
TB2	628.60(1%)	220.01	30.17
6 NTB	1026.33	373.58	20.53
TB1	1002.70 (2.3%)	359.97	10.03
TB2	1019.95(1%)	367.18	20.44
8 NTB	1418.75	517.84	10.64
TB1	1401.30 (1.3%)	508.27	2.40
TB2	1411.00(1%)	509.37	10.70

The results of Table 1 are obtained by taking the average of 10 similar runs of the benchmark. The system and wait for IO times are computed by multiplying the sar results of %sys and %wio by the elapsed time. The value in the parenthesis is the percentage improvement of that

configuration relative to no text bit setting (NTB).

It can be seen that the recommended settings by the text bit program results in shorter elapsed time(1-4%) for the benchmark. Although this is a small improvement it should be noted that this tuning operation requires little effort and cost. The rate of improvement decreases as number of scripts increases. This is expected since with several terminals running, the probability of having the command in the main memory increases.

### 2.2.2 Dcopy Incentive Module

This module estimates the improvement in disk response time if a file system on that drive is reorganized using dcopy (dcopy(1M)).

A UNIX file system is created in such a way that the list of free disk blocks are ordered with an optimal gap between every two consecutive free blocks. Assuming two consecutive free list blocks are allocated to two consecutive blocks of a file, the gap ensures that with sequential access to the file, the seek and latency times are minimal. However, the physical layout of files in a file system deteriorate as blocks are added to and deleted from the free list.

The dcopy command reorganizes a file system, making sure that all consecutive blocks of files are placed within the optimal gap on the disk (if possible). Furthermore, it compacts the directory blocks to remove null entries and reduce the size.

File system reorganization can improve the computer system's response time considerably in environments where the files are being used sequentially and disk is a bottleneck resource. However, it may not produce a significant improvement if the file system is relatively organized or it is being used during the less busy periods.

The dcopy operation is time consuming. Also, the file system being reorganized is not usable (should be unmounted) during the dcopy operation. In a dynamic environment, a dcopied file system may not maintain its organization long enough to make up for the long running time of the dcopy operation itself. For these reasons and also due to the fact that the improvements are difficult to quantify, system administrators hesitate to perform a dcopy.

The dcopy incentive module provides a measure of improvement in disk response time if the file system is reorganized. It gives the user an idea of how necessary it is to perform a dcopy. To the Tuning Aid, it provides the information of a possible remedy for slow IO operation. Furthermore, by maintaining a history of the rate of change of the file systems (in terms of organization), it can also estimate the rate at which the file system physical layout on the drive becomes random.

The analysis requires an enumeration of all the blocks in the file system. A suitable place to put the analysis is in the fsck (fsck(1M)) program. The fsck routine checks the file system for consistency and to do that, it has to look at all the file system blocks. The added analysis uses the file-block information from the fsck program and computes an approximate improvement in disk response time using a disk performance model. The model receives as its input the disk profile data (sadb(1M)) and system activity report (sar(1)). The modified fsck program provides a measure of how unorganized the file system is, in terms of expected disk response time improvements after dcopy. Furthermore, it computes the extra number of directory blocks occupied by null directory entries.

An example of an action which is added to the Tuning Aid rules is the following:

```
if (disk_load is HIGH) then
    check cache hit ratios
    check file system organization
```

The following is an example of the information provided by the Tuning Aid using this module.

if (file system *X* is reorganized) then  
5% improvements in disk response time is predicted and  
10 directory blocks are released and  
the expected time to reach the present state is 30 days;

The last observation refers to how fast the file system is expected to reach the present state in terms of access inefficiency.

#### 2.2.2.1 Experimental Results

This module was tested in controlled environment on an AT&T 3B2/400. The benchmark consists of a number of IO-bound jobs using a file system that had not been recently organized. The system activity measurement (sar) and the disk activity profile (sadb) during the benchmark period were recorded. Using the disk activity profile data from the benchmark, the dcopy incentive module was run on the file system. The result of this program is given in Figure 3.

```
/dev/dsk/c1d0s2 (NO WRITE)
File System: usr1 Volume: hokai2

** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
Saving in average access time if dcopy is run: 4.40 ms
(15% reduction in disk utilization).
Number of null directory blocks: 1

** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Free List

3784 files 49200 blocks 38504 free
```

**Figure 3.** Result of the Dcopy Incentive Module on a File System.

As can be seen, the module has been included in the fsck program. The module gives the total number of blocks between every two consecutive blocks in excess of the optimal gap. It also provides the total number of cylinders between two consecutive blocks. These numbers (not shown in the figure) by themselves are not very useful. If a file is hardly ever used, the efficiency of access to that file is of little importance. In the figure, we also see a prediction for disk response time improvements given a file system reorganization. This number is used as a measure for randomness of the file system's physical layout. Its computation takes into account the access frequency to the particular cylinders. Also in the figure the number of empty directory blocks that can be saved if file system is reorganized is reported.

The file system is then reorganized using dcopy. The same benchmark is run on the two file systems. The sar results of the two benchmark runs appear in Figure 4. Entries for hdsk-0 and hdsk-1 give the disk activity data for the two drives with the unorganized and the organized file systems respectively. A 20% improvement in disk utilization was achieved. The predicted improvement by the module was 15%.

device	%busy	avque	r+w/s	blks/s	avwait
hdsk-0	60	7.2	20	40	184.9
hdsk-1	48	5.0	20	40	96.3

Figure 4. Sar Results for the Disk Activities.

### 2.2.3 Disk Buffer Sizing Module

In the UNIX operating system, a number of buffers are reserved in the main memory for the file system IO operations. Once a request for a block is issued, the block IO subsystem searches this buffer pool for the block. If it is not found, then the block is copied from the disk into the buffer pool. As a result, the previous content of a buffer disappears. The buffer that is being used for this purpose is the least recently used buffer in the buffer pool (approximately).

The idea of the buffer pool is to reduce disk access by reusing blocks that are available in the pool and hence speedup the IO operations to satisfy cpu requirements. Usually, the larger the number of buffers, the higher the probability of finding a block in the main memory. On the other hand, adding buffers to the buffer pool takes the space away from the user memory which can cause increased swapping or paging.

The number of the buffers in the buffer pool is a tunable parameter. It is suggested that if the hit ratios (the ratio of the number of blocks found in buffer pool to number of blocks requested), are low and the memory is not a scarce resource, then the number of buffers be increased. However, it is not clear how much increase is needed, or, if the increase in number of buffers will result in any improvements.

To analyze the effects of buffer pool size on the system performance, a module has been developed which estimates reduction(s) in the number of IO transfers to/from the disk, given the buffer pool is increased to certain level(s). The reduction(s) in number of physical IO can be translated into reductions in disk utilization.

This module has been added to the source of the UNIX operating system. The system runs with the regular number of disk buffers while effects of some extra buffers (shadow buffers) are being simulated using the real system's data. The modified kernel is expected to run for the duration of measurements. In fact, it can be part of the kernel which is activated on request. The system activity report (sar(1)) was also modified to reflect the new measurement data, that is, the predicted reductions in disk IO operations given the buffer size is increased.

The module provides necessary information to the rule base, when an increase in buffer cache size seems to be the solution to an IO problem. It can suggest the amount of increase, and also the improvements. For example the following rule can be added to the Tuning Aid:

if (the level 1 program recommends increasing the number of buffers) then  
analyze the results of the buffer analysis module;

#### 2.2.3.1 Experimental Results

This module was tested in a controlled environment using a benchmark similar to our previous experiments (text bit). Different number of scripts were run on a number of UNIX kernels with and without the disk buffer sizing module. Table 2 gives the results of these experiments. In this table, the same benchmark was run on three kernels with different configurations. *R1* refers to the original kernel (without the analysis code) with *x* number of system buffers. *x* is 768 in the table. *P* refers to the kernel that includes the analysis and is configured with *y* system buffers and *z* shadow

buffers where  $y+z=x$ . In this experiment  $y$  and  $z$  are 256 and 512 respectively. Finally,  $R2$  refers to the original kernel with  $y$  (256) number of system buffers.

TABLE 2. Real System vs. Predictions.

		real buffer,hash		shadow buffer,hash			
Real system (R1)		768,128					
Predictions (P)		256,64		512,128			
Real system (R2)		256,64					
No. of Scripts	No. of Phys. IO	No. of Logical IO	Predicted Phys. IO	Cache Hit Ratio	Elapsed Time	Ovhd	
1 R1	892	19422	-	95.4	107.55	0.5%	
P	2207	19760	827	88.8->95.8	130.90		
R2	2200	19757	-	88.9	130.20		
4 R1	5674	79727	-	92.9	369.35	1.1%	
P	10300	80552	5464	87.2->93.2	402.30		
R2	10261	80941	-	87.3	397.90		
8 R1	12711	157844	-	91.9	729.06	2%	
P	21371	159806	12826	86.6->92	789.00		
R2	20634	159347	-	87.1	773.80		
12 R1	20439	235055	-	91.3	1093.20	1.2%	
P	32722	237661	20559	86.2->91.3	1172.40		
R2	31722	237355	-	86.6	1159.00		

The comparison is made between the two sets of data from  $R1$  and  $P$  for the validity of the predictions produced by  $P$ . Note that  $P$  results should tell us: "If in a system with  $y$  buffers the number of buffers is increased by  $z$ , then the number of physical IO accesses (actual number of requests to the disk drive) is predicted to be  $p_{io}$  (column 4 of the table)."  $P_{io}$  is then compared to the actual result obtained from the  $R1$  kernel (column 2,  $R1$  row).

The two sets of data from  $P$  and  $R2$  are compared to estimate the overhead of adding the module to the source code. Note that both configurations have the same number of system buffers, while  $P$  has the extra shadow buffer code.

Similar to the previous benchmark runs, the elapsed times in the table are the time periods during which the benchmark scripts were running. The other entries in the table are obtained by `sar` (option -b) measurements during the benchmark runs. The values appearing in the tables are averaged over 15 runs of the same benchmark on the same configuration.

It can be seen that the predictions are reasonably accurate when we compare the total number of IO operations. For example, in the first row  $R1$  in Table 2 the total number of physical IO operations is 892. The same activity for  $P$  is 2207 and the predicted number given the number of buffers are raised from 256 to 768 is 827 (4th. column). The predictions when translated into cache hit ratios are even closer to the real system. The reason is that the difference between predictions and the actual results are small relative to the total number of IO activities that have occurred (3rd. column).

The overhead of the extra code added to the kernel code was measured in terms of percentage of increase in the total elapsed time of any benchmark run. This variable ranged over 0.5% to 2%.

In these experiments we have only simulated effects of one level increase to the system buffers. That is, the module can only tell us what will happen if we add  $y$  buffers to the system buffers. The program should be able to predict what happens if buffers are increased by  $y_1, y_2, y_3$ , etc. Presently, this feature is under development.

Two other issues in this area remain to be investigated. One is the effect of reducing number of buffers on the performance, and the other is the effect of buffer pool adjustment on the memory. At the moment to deal with the latter issue, Tuning Aid considers the swap out ratios (from sar) as a measure for memory requirements. This value does not indicate what will happen if less memory is provided, it can only tell us if, with the current configuration, the memory is a scarce resource or not.

#### *2.2.4 File System Allocation Module*

Major performance improvements can be obtained by placing file systems on drives in such a way that a) the more frequently used file systems are placed on neighboring partitions on the drive and b) the load on the drives are relatively balanced.

In this part, a heuristic method is developed which produces several configurations and selects the one that minimizes some cost. The heuristic tries to meet the two objectives (a) and (b) above while satisfying the operational constraints. The idea is to generate many feasible configurations and test the effectiveness of every configuration by running it through a cost model. The configuration that is considered optimal within those tested is the one that gives the minimum of the maximum response time experienced by requests to a drive or a file system during periods of heavy disk utilization.

An example of an added rule to the rule base is as follows:

if (a disk is highly utilized and the cache hit ratios are high) then  
    (i) check file systems' organization  
    (ii) check file systems' configuration

In (i) above, Tuning Aid looks into the possibility of improving a slow IO condition by organizing the file systems. In (ii), the file systems' position on the drives and partitions are being reexamined.

The input to this module is the data from the system activity disk profile and system activity report. Clearly, the recommended configuration depends on the type of activity during the measurement period. Different sets of input data from different time periods may result in different suggestions. In such cases, Tuning Aid presents all possibilities and conditions under which a particular configuration is suggested. The user needs to make the final decision.

##### *2.2.4.1 Experimental Results*

To show the validity of this module, experiments were performed in a controlled environment on an AT&T 3B2/400 with two disk drives. The experiments consisted of running different programs using one or more file systems. The disk activity profile during the experiment was recorded and then used to run the file system allocation module. The file systems are then configured as suggested by the module and the same benchmark was run on the new configuration. System activity data during the two experiments is shown in Figure 5. As can be seen the disk utilization for the two drives changed from 79% and 5% to 32% and 51%. The model predicted utilizations of 28% and 48% after the change.

#### *2.3 Level 3*

At this level, one of the major sources of information to the Tuning Aid is history. The measurement data can be noisy and time varying, the models although sound, give approximate

Before the change:

device	%busy	%avque	r+w/s	blks/s	avwait	avserv
hdsk-0	79	5.7	30	60	124.1	26.3
hdsk-1	5	1.6	2	3	17.2	29.0

After the change:

device	%busy	%avque	r+w/s	blks/s	avwait	avserv
hdsk-0	32	2.5	13	28	34.2	25.9
hdsk-1	51	3.5	18	34	69.4	28.3

**Figure 5.** Disk Performance Data on the two Configurations.

results, and some threshold values are too conservative to detect subtle problem areas. To fill these gaps and provide more information, the system has to learn from its history. The results of previous tunings and the performance data in the past (with similar workload) can provide substantial information to the system. The process of filtering and analyzing history information in a way is a learning process for the Tuning Aid. It can adjust its threshold values and include parameters in its decision making that have not been considered before. For example, the history might show that a certain condition occurs periodically for a short time. The Tuning Aid can then point out that the observations during this period are not typical and the user need not be alarmed, i.e., it may not be necessary to make an investment of purchasing new equipment in order to deal with problems that occur infrequently.

At the present time, little work has been done in this area. The disk buffer sizing module in a sense relies on the history information by accumulating system performance data for different periods. The dcopy incentive module accumulates the rate at which the file systems become unorganized and predicts the rate of decrease in efficiency in future. Nevertheless, learning and modifying decisions dynamically are not included in the system. This is an area which needs further research.

### 3. Conclusions

The Tuning Aid is being tested on several public systems. The feedback we have received from users is positive. It can be observed that such tools not only help less experienced operators and administrators but also the expert system administrators by extracting the useful information from the large volume of data and providing understandable performance measures about their systems.

Another major observation which resulted from this work is that to succeed in tuning, simple rules based on thresholds are not sufficient. We can do much better by complementing the expert's knowledge with quantitative techniques and in some cases models.

To enhance the capabilities of Tuning Aid, much work remains to be done. Future efforts will concentrate on increasing the learning power (level 3) of the tool and adding modules which are able to tune other parameters.

### Acknowledgements

I would like to thank Robert Morris for proposing this project, contributing ideas and supporting this work. Also, I would like to thank Y.T. Wang and Danny Chen for the comments and time for discussions.

### REFERENCES

- [ART 85] Artis, H.P., "Using Expert Systems for Analyzing RMF Data," International Conference on the Management and Performance Evaluation of Computer Systems, Dallas 1985.
- [HEV 85] Hellerstein, J. and Van Woerkom, H., "YSCOPE: A Shell for Building Expert Systems for Solving Computer-Performance Problems," International Conference on the Management and Performance Evaluation of Computer Systems, Dallas 1985.
- [HSI 86] Hsiao, J., "A System for Interpreting Sar Data," in preparation.
- [KAR 84] Karnaugh, M., Ennis, R.L., Griesmer, S.L., Hong, S.J., Klein, D.A., Milliken, K.R., Schor, M.I. and Van Woerkom, H.M., "A Computer Operator's Expert System," International Conference on Computer Communications, Sydney, Oct. 1984, pp. 812-817.
- [LEV 85] Levine, A.P., "ESP: An Expert System for Computer Performance Management," International Conference on the Management and Performance Evaluation of Computer Systems, Dallas 1985.
- [SBD 85] Stroebel, G.J., Baxter, R.D. and Denny, M.J., "A Capacity Planning Expert System for IBM System/38," International Conference on the Management and Performance Evaluation of Computer Systems, Dallas 1985.
- [UNI 84] "UNIX System V - Release 2.0, Tuning and Configuration Guide," 307-121, Issue 1, 1984.
- [UNI 85] "AT&T 3B2 Computer, UNIX System V - Release 2, System Administrator's Guide," Chapter 6, 305-477, Issue 1, 1985.

# Software Performance Analysis Using Call Graphs and Workstation Graphics

David B. Leblang  
Apollo Computer  
Chelmsford, Ma. 01824  
leblang@apollo.uucp

## Abstract

This paper describes a new workstation tool for software performance profiling. **DPAT** (the DOMAIN® Performance Analysis Tool®) combines workstation graphics, comprehensive program monitoring techniques, and a powerful, interactive, data analysis tool. **DPAT** measures the performance of a program, including I/O, paging, and system calls, at the procedure level. **DPAT** has many significant advantages over existing tools like **gprof**[Gra83] and **prof**[UPM] including the use of graphics, a much more accurate model of program execution, and a full sample history, which enable the user to replay the program's execution and to produce a variety of reports.

**DPAT** has three modes: *monitor*, *analyze*, and *playback*. During a monitoring session, as the program executes, **DPAT** graphically displays the current call stack and produces bar charts showing the most active procedures. **DPAT's** *playback* mode graphically reviews the execution of a monitored program in *forward* or *reverse* by means of a "tape recorder" on which the user may click forward, reverse, fast-forward, etc. Playback mode can also draw flow diagrams of the program's dynamic structure. This playback capability is possible because **DPAT** records, in a very dense format, every sample taken, rather than just a running summary of samples. The full sample history also enables **DPAT** to answer contextual questions about the program, for example, how much of a compiler's time is spent in the symbol table package when the package is *directly* or *indirectly* called from the parser.

**DPAT's** *analyze* mode produces reports based on the information gathered by the monitor mode and stored in a datafile. A *ranked* report lists procedures in sorted order based on the percentage of time consumed. A *hierarchical* report is organized according to a top-down dynamic decomposition of the target program. Both types of reports reflect time and page faults spent *in-or-under* each procedure (ie. the time required to execute the abstraction regardless of how the abstraction is implemented). *Analyze* mode allows the user to re-analyze a datafile focusing on particular parts of the target program.

## Background

Most performance analyzers use *counters* or *statistical sampling* to monitor program performance. *Counters* embedded in source code by the programmer or in the executable code by a compiler count the number of times a procedure is entered. Counters usually cause only a small perturbation in the program's execution and can be used to accurately determine whether or not a procedure was executed (which is useful in a quality assurance/test coverage context). However, the number of times a procedure was executed does not necessarily correspond to the *time* spent in the procedure and so counters alone are not sufficient for an accurate program profiler.

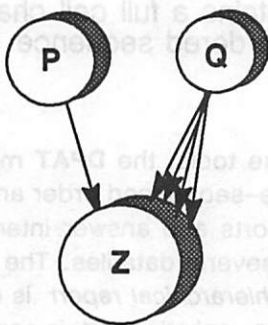
A *statistical profiler* periodically interrupts a program and gathers information about the program's current execution location. The UNIX® performance analysis tool, **prof**, is a statistical

sampler that provide basic performance analysis support for UNIX systems. **Prof** periodically records the PC (program counter) of the target program and then prints a relative frequency chart at the procedure level after the program has completed. **Prof** is good at finding time-consuming program sections that result from compute-intensive code, as long as those sections make no procedure calls. Unfortunately most modern programs use many procedure calls to implement abstractions. *Their performance problems are more related to the algorithms used than to the particular code sequence used to implement the algorithm.* **Prof** is poor at analyzing highly modular code with many calls because these programs have no "hot spot" in which a particular PC appears repeatedly. Rather, the PC is distributed among many procedures that implement a particular abstraction. Some tools, like the VAX/VMS Performance and Coverage Analyzer [VAX85] use both sampling and counters to produce reports similar to **prof** and also test coverage reports.

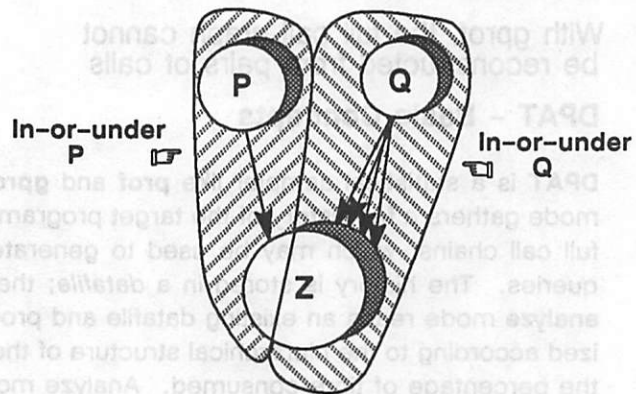
In order to gain meaningful performance statistics on structured programs the performance analysis tool must understand the hierarchical nature of the program. **Gprof** was developed to address this issue; Graham [Gra83] discusses the design of **gprof** and the motivations for call graph based profiling in detail. **Gprof** works by using a combination of counters and statistical sampling. **Gprof** determines the time spent *strictly in* a procedure the same way **prof** does (ie. based on statistical samples of the PC). **Gprof** also maintains execution counts for each procedure call. These counts require compiler support so a program must be specially compiled to use **gprof** (**prof** also requires special compilation). In addition, **gprof** constructs a call graph of the target program by recording caller/callee pairs at procedure entry (see [Gra83]).

**Gprof** determines the time spent *in-or-under* a procedure P by summing up the time spent *strictly in* the procedure and a *portion of the time* spent in procedures called *directly or indirectly* by P. The *portion* of a callee charged to the caller is based on the execution counts.

% time spent directly in procedure		number of calls from -> to		% time spent in-or-under procedure	
P	30%	P -> Z	1	P	40% (30% in P + 1/5 of 50% in Z)
Q	20%	Q -> Z	4	Q	60% (20% in Q + 4/5 of 50% in Z)
Z	50%			Z	50% (50% in Z + no calls out)



Raw Data gathered by gprof



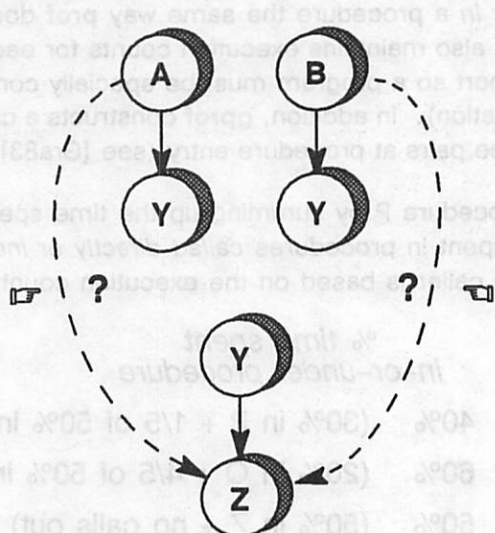
Conclusions reached by gprof

In the example above, P calls Z one time while Q calls Z 4 times. Although Q itself only required 20% of the execution time the *abstraction* implemented by Q (time in-or-under Q) required 60% of the time, because Q makes such heavy use of Z. Although procedure P appears to be more expensive than procedure Q when considered alone, it is really procedure

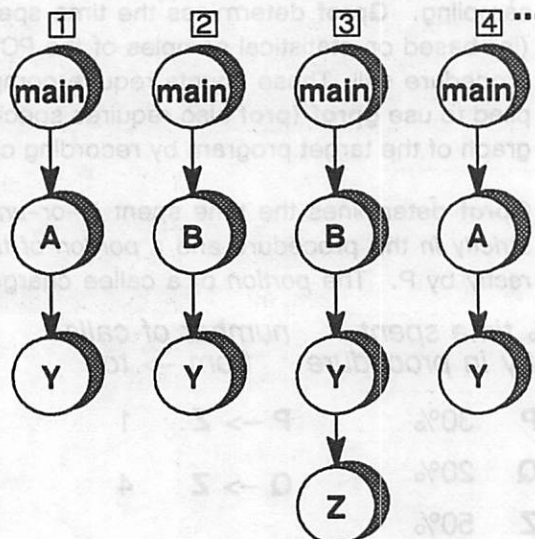
Q that is more expensive (P 40% vs. Q 60%) when the full call graph is considered. Note that since **prof** only looks at the time *in* a procedure it would have claimed the reverse (P 30% vs. Q 20% since **prof** assigns frequencies strictly on where the program counter is found when the sample is taken).

This hierarchical view of execution time is very valuable. Unfortunately, the view provided by **gprof** is not likely to be accurate because **gprof** makes the assumption that execution counts can be used to apportion the time spent in the called procedures. In the example above, suppose Z is a string copy procedure and Q calls it several times to copy short strings but P calls it once to copy a very long string. Assigning P and Q portions of Z's time based on just the number of calls would be inaccurate.

A related problem occurs when multiple levels of procedure calls are involved. Since **gprof** only records caller/callee pairs it cannot answer contextual questions about the program that involve more than one level of calls. In the example below, A and B both call Y, and Y sometimes calls Z. **Gprof** always charges some of Z's time to A even though A's calls to Y *never* resulted in calls to Z.



With **gprof**, the full call graph cannot be reconstructed from pairs of calls



DPAT maintains a full call chain in time ordered sequence

### DPAT - Basic Concepts

DPAT is a statistical sampler like **prof** and **gprof**, but unlike those tools, the **DPAT monitor** mode gathers a full history of the target program execution, in time-sequenced order and with full call chains, which may be used to generate a variety of reports and answer interactive queries. The history is stored in a *datafile*; the user may have several datafiles. The **DPAT analyze** mode reads an existing datafile and produces reports. A *hierarchical report* is organized according to the hierarchical structure of the target program; a *ranked report* is sorted by the percentage of time consumed. Analyze mode is interactive and allows the user to focus DPAT's analysis on particular parts of the target program.

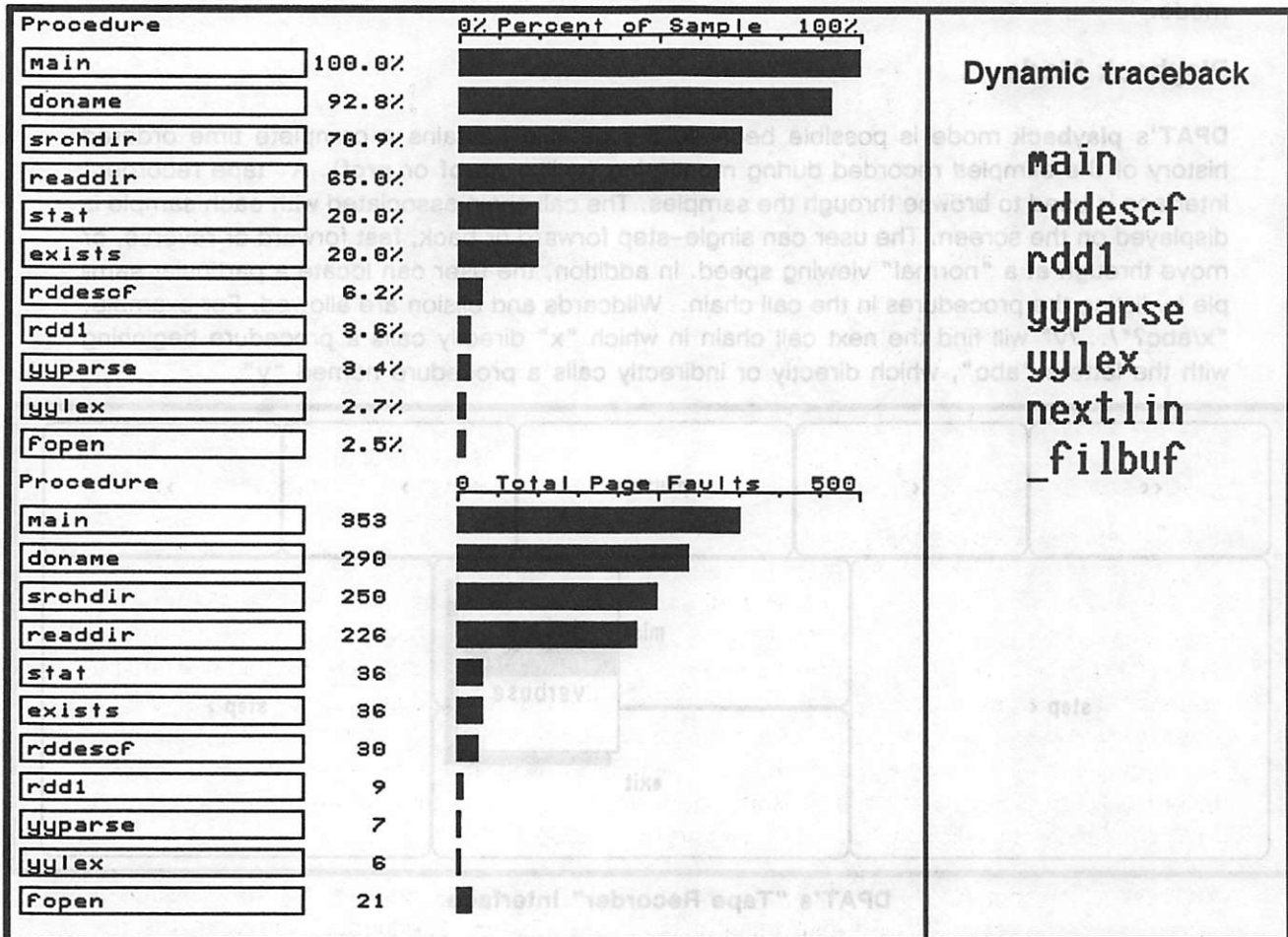
DPAT uses workstation graphics for its user interface (icons and menus), for the continuous displays of a bar chart of the most active procedures, for the continuous display of the dynamic call stack, and to draw flow graphs of the target program. Since the tips of the call stack may flash by very quickly when sampling, the **DPAT playback** mode allows the user to review

the samples gathered. In playback mode the user can review the program execution in a forward or reverse direction and at varying speeds. This mode also allows direct access to a particular sample, and searching by procedure names.

Rather than just incrementing a counter for each PC sampled (like `prof[UPM]`), or for each PC and return address (like `gprof[Gra83]`), each DPAT sample records the full call chain leading to the current PC. These samples are kept in a time-ordered list, which allows DPAT to estimate the amount of time each procedure is **active** (that is, the relative amount of time spent either in the procedure itself or in the procedures it calls directly or indirectly). Therefore, DPAT can identify procedures that consume a relatively large amount of time, *regardless* of whether those procedures consume time directly, or invoke other time-consuming procedures or system services.

### Monitor Mode

DPAT monitors a target program from a separate process and when necessary, reads the address space of the target program (like a cross-process debugger using `ptrace`). The cross-process operation of DPAT minimizes the effect on the target program and means that users need not specially compile the target program in order to analyze it. It also means that a program that is already running (like a server) can be analyzed. DPAT may be used with the debugger to start analyzing a section of a program at a breakpoint and stop monitoring it when the next breakpoint is encountered. This is especially useful with very large programs.



Screen Image of a DPAT monitoring session

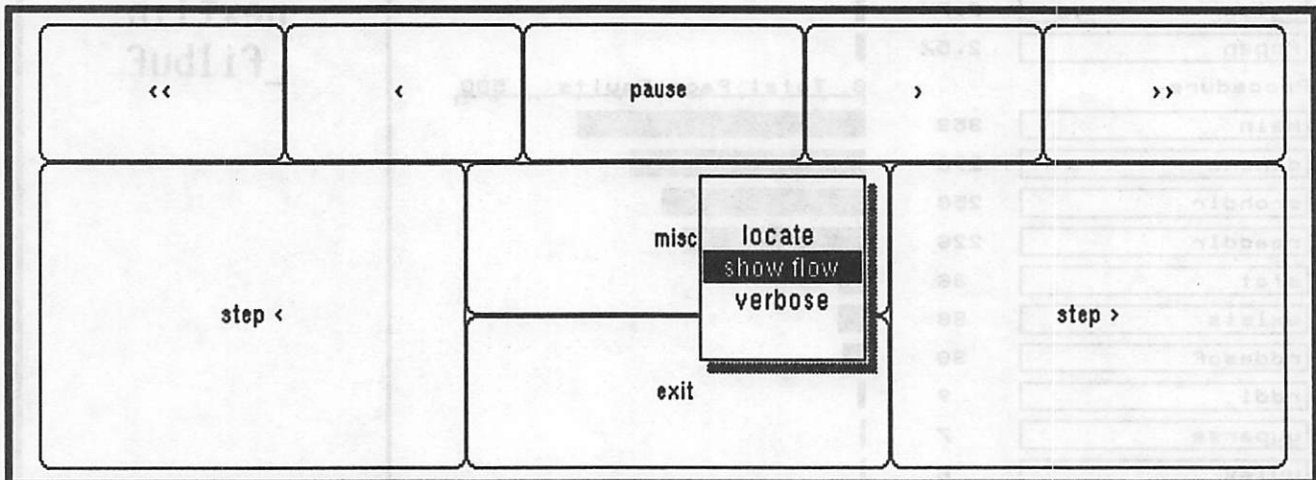
DPAT's basic operation is to periodically suspend the target process, read the stack and record the procedure call chain, resume the target process, and then sleep until the next sample. For each sample, DPAT writes the call chain to a *datafile* and updates the graphic displays. To keep the datafile small, DPAT stores indexes into a symbol table for each procedure, rather than the full name of the procedure. When the monitoring phase is complete, DPAT writes the symbol table to the datafile.

Since it monitors from a separate process, DPAT is able to detect I/O and paging-bound procedures by continuing to sample the target process even when the target process is blocked on an I/O operation, page fault, or system call. A target program procedure waiting on I/O transfers or paging will appear in several samples while these operations are being processed. The time consumed by such a procedure will be properly accounted for in later analysis. Prof and gprof rely on the target to interrupt itself in order to take samples, so the time consumed by non-interruptible operations (like many kernel calls and types of I/O) is missed.

The default sampling rate is 50 milliseconds. The user can decrease this rate to 1 millisecond, or increase it to several seconds. Due to sampling overhead the shorter the sample interval the more real time is required to complete the monitoring of the target program. Since the target program is suspended briefly during sampling, the effect on its execution is minimized. Also, since displaying bar charts and call stacks increases sampling overhead, DPAT allows the user to turn off these displays during monitoring and review them later in the playback mode.

## Playback Mode

DPAT's playback mode is possible because the datafile contains a complete time ordered history of the samples recorded during monitoring (unlike gprof or prof). A "tape recorder" interface is used to browse through the samples. The call chain associated with each sample is displayed on the screen. The user can single-step forward or back, fast forward or reverse, or move through at a "normal" viewing speed. In addition, the user can locate a particular sample by listing the procedures in the call chain. Wildcards and elision are allowed. For example, "x/abc?\*/.../y" will find the next call chain in which "x" directly calls a procedure beginning with the letters "abc", which directly or indirectly calls a procedure named "y".

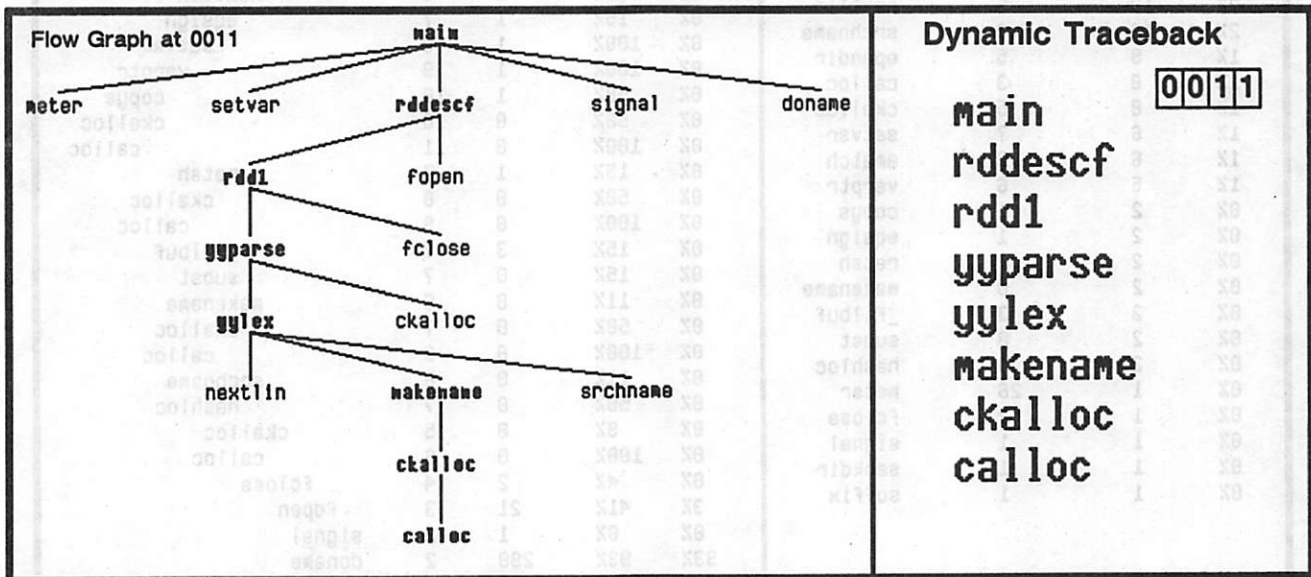


DPAT's "Tape Recorder" Interface

Playback's **show flow** option draws a flow graph of the target program based on the calling structure indicated by the samples recorded in the datafile. Each flow graph is drawn around a

particular call chain — that is, DPAT draws a slice of the full flow graph that includes the call chain requested. At each level in the graph the active procedure (e.g. the one in the selected call chain) is shown in bold and other procedures that are called from the same parent procedure are shown. Procedures are shown left to right in the order in which they were called. Flow graphs are very helpful in understanding the organization of a program.

DPAT only graphs the immediate “neighborhood” of the call chain because a full flow graph of most non-trivial programs would not fit on the screen (or would be incomprehensible if it did). The hierarchical report produced by DPAT’s **analyze** mode prints the full calling tree in the form of indented text.



An Example of a Flow Graph Display

## DPAT as a Debugging Aid

DPAT can also be used as a debugging aid. For example, if a program is running for several minutes with no output, DPAT can gain control of it from another window and display the call stack dynamically enabling the user to see what the program is doing. Or, DPAT can watch a program that runs for a while and then faults. A post-mortem debugger may be able to tell you where the fault occurred but not the events leading up to the fault. DPAT can review the datafile in reverse order (from the fault backwards) to see the call sequences that led to the fault.

## Analyze Mode

The **analyze** mode in DPAT analyzes the raw data stored in a datafile. Analyze mode produces a summary report giving the percentage of time that each procedure was active and the number of page faults that occurred while each procedure was active. Analyze mode is an iterative process; since the execution data has been captured in a datafile, DPAT can generate a variety of reports depending on the parameters specified for the analysis. These parameters (described below) are used to emphasize or isolate areas of interest.

DPAT can produce two major types of reports: a *hierarchical report* which is organized according to the hierarchical structure of the target program, and a *ranked report*, which is sorted by percentage of time consumed. Both types of reports are subject to *filtering* and *grouping*, which are described below.

%	Samples	Page Faults	Routine	%total	%parent	page faults	level	
100%	658	353	main	100%	100%	353	1	main
93%	611	290	doname	0%	0%	26	2	meter
71%	467	250	srchdir	1%	1%	6	2	setvar
65%	428	226	readdir	0%	75%	5	3	varptr
20%	132	36	stat	0%	100%	5	4	ckalloc
20%	132	36	exists	0%	67%	3	5	calloc
6%	41	30	rddescf	6%	6%	30	2	rddescf
4%	24	9	rdd1	4%	59%	9	3	rdd1
3%	23	7	yyparse	3%	96%	7	4	yyparse
3%	18	6	yylex	3%	78%	6	5	yylex
3%	17	21	fopen	2%	72%	5	6	nextlin
2%	13	5	nextlin	0%	15%	1	7	eqsign
2%	10	3	srchname	0%	100%	1	8	setvar
1%	9	5	opendir	0%	100%	1	9	varptr
1%	8	3	calloc	0%	50%	1	10	copys
1%	8	5	ckalloc	0%	50%	0	10	ckalloc
1%	6	7	setvar	0%	100%	0	11	calloc
1%	6	3	amatch	0%	15%	1	7	retsh
1%	5	6	varptr	0%	50%	0	8	ckalloc
0%	2	1	copys	0%	100%	0	9	calloc
0%	2	1	eqsign	0%	15%	3	7	_filbuf
0%	2	1	retsh	0%	15%	0	7	subst
0%	2	0	makename	0%	11%	0	6	makename
0%	2	3	_filbuf	0%	50%	0	7	ckalloc
0%	2	0	subst	0%	100%	0	8	calloc
0%	2	0	hashloc	0%	11%	0	6	srchname
0%	1	26	meter	0%	50%	0	7	hashloc
0%	1	2	fclose	0%	9%	0	5	ckalloc
0%	1	1	signal	0%	100%	0	6	calloc
0%	1	1	seekdir	0%	4%	2	4	fclose
0%	1	1	suffix	3%	41%	21	3	fopen
				0%	0%	1	2	signal
				93%	93%	290	2	doname
				20%	22%	36	3	exists
				20%	100%	36	4	stat
							3	doname ...
				71%	76%	250	3	srchdir
				1%	2%	5	4	opendir
				65%	92%	226	4	readdir
				0%	0%	1	4	seekdir
				1%	1%	3	4	amatch
							5	amatch ...
				0%	0%	0	4	copys
				0%	100%	0	5	calloc
				1%	1%	3	3	srchname
				0%	13%	0	4	hashloc
				0%	0%	1	3	suffix

Ranked Report

Hierarchical Report

## Focusing Reports Using Filters and Groups

**Filters** limit the scope of a ranked or hierarchical report by specifying a subset of samples for analysis. Samples that satisfy the filter are included in the summary report; all others are excluded. Percentages are calculated relative to the number of samples that pass the filter. A filter is specified with a *call chain sequence* or with a sample number. The call chain sequence may include wildcards and elision.

**After** and **Until** filters can be used to select a range of samples. An **after** filter specifies the first sample in the datafile to be used; an **until** filter specifies the final sample to be used. For example, you could specify **after** "flow" and **until** "peephole" in order to limit analysis to the

middle phases of a compiler (e.g. flow analysis and code generation but not syntax, peephole optimizer, or object code generator). **Matching** filters specify a pattern that is applied to all sample call chains. Only those samples matching the pattern are analyzed. Matching filters are especially useful for limiting reports to areas of interest -- for example, a matching filter of "yparse" can be used to focus on the YACC generated parser. A more complex filter, such as "yparse/.../str?\*\"", could focus on the parser's direct or indirect usage of string functions.

The **DPAT cut off percentage** filter eliminates low percentage entries in ranked or hierarchical reports. The **cut off level** filter eliminates deeply-nested entries from a hierarchical report. These filters generally result in significantly smaller reports focused on the most time-consuming procedures.

**DPAT groups** combine related samples into a single report entry. For example, the group "?\*printf" would combine into one entry all the time spent in-or-under the i/o formatting functions (eg. printf, sprintf, fprintf). In addition, groups combine all procedures below a grouped procedure into a single entry. For example, the group "yparse" would group all of the YACC parser into a single entry. An alternate name is given for the group in the report (eg. YACC\_PARSER for the group "yparse").

## Handling Recursion

In the call chain for a recursive procedure, the same procedure occurs more than once. To avoid double counting these procedures when computing statistics, the analyze phase must detect recursion. Hierarchical reports indicate recursion by printing the procedure name followed by "...". The hierarchical report above shows that the procedure "doname" and "amatch" are recursive.

**DPAT** detects recursion by maintaining a *ticket counter* in the **DPAT** symbol table for each procedure. The ticket indicates the last sample in which this procedure appeared. The tickets are zeroed at the start of the analysis for each report. They are set to the current sample number as the datafile is scanned to produce the report. Before counting a procedure, **DPAT** checks the symbol table to see if this procedure has already appeared during the current sample number; if so, the procedure is not counted again.

## Conclusions

**DPAT** has proven to be an extremely helpful and accurate tool. It is especially good at finding inefficient I/O usage, poor locality of reference (resulting in a lot of page faults), and expensive algorithms. **DPAT** is also valuable in capturing the sequence of events that lead to catastrophic program failures (eg. stack corruption) for which post-mortem debuggers are ineffective. Finding and fixing performance problems is not always easy, but **DPAT** has proven to be of considerable help.

## Current Status

**DPAT** is a commercially-available as part of the DOMAIN Performance Analysis Kit<sup>®</sup>, a product for Apollo workstations. It runs on Apollo's DOMAIN/IX distributed UNIX environment. Other tools in the DOMAIN Performance Analysis Kit<sup>®</sup> include **HPC** and **DSPST**. **HPC** is a very high speed PC histogram profiler similar to **prof** except it can profile a program at the single statement level. **DSPST** graphically displays system resource usage (I/O, network transfers, etc.) and the relative CPU time used by all processes running on a workstation.

## References

- [Gra83] S. Graham, P. Kessler, M. McKusick  
"An execution profiler for modular programs"  
Software Practice and Experience, 13, 8 (Aug-83).
- [UPM] Unix Programmer's Manual  
Bell Labs, Murray Hill, N.J.
- [VAX85] VAX Performance and Test Coverage Analyzer,  
DEC order number AA-EB54B-TE, Oct. 1985

## Handling Recursion

In the call chain for a recursive procedure, the same procedure occurs more than once. To avoid double counting these procedures when computing statistics, the analyze phase must detect recursion. Hierarchical reports indicate recursion by phrasing the procedure name followed by "...". The hierarchical report above shows that the procedure "dname" and "smatch" are recursive.

DPAT detects recursion by maintaining a ticket counter in the DPAT symbol table for each procedure. The ticket indicates the last sample in which this procedure appeared. The tickets are zeroed at the start of the analysis for each report. They are set to the current sample number as the database is scanned to produce the report. Before counting a procedure, DPAT checks the symbol table to see if this procedure has already appeared during the current sample number; if so, the procedure is not counted again.

## Conclusions

DPAT has proven to be an extremely helpful and accurate tool. It is especially good at finding inefficient I/O usage, poor locality of reference (resulting in a lot of page faults), and expensive algorithms. DPAT is also valuable in capturing the sequence of events that lead to catastrophic program failures (eg. stack corruption) for which post-mortem debuggers are ineffective. Finding and fixing performance problems is not always easy, but DPAT has proven to be of considerable help.

## Current Status

DPAT is a commercially-available as part of the DOMAIN Performance Analysis Kit<sup>®</sup>, a product for Apollo workstations. It runs on Apollo's DOMAINIX distributed UNIX environment. Other tools in the DOMAIN Performance Analysis Kit<sup>®</sup> include HPC and DSPST. HPC is a very high speed PC histogram profiler similar to prof except it can profile a program at the single statement level. DSPST graphically displays system resource usage (I/O, network transfers, etc.) and the relative CPU time used by all processes running on a workstation.

# ILMON: A UNIX Network Monitoring Facility

Lewis Barnett

Department of Computer Sciences  
Taylor Hall 2.124  
University of Texas at Austin  
Austin, TX 78712

Michael K. Molloy \*

Department of Computer Science  
4212 Wean Hall  
Carnegie-Mellon University  
Pittsburgh, PA, 15213

## ABSTRACT

This report describes ILMON, a network monitoring facility running under the Berkeley UNIX<sup>†</sup> (4.3BSD) operating system. ILMON is composed of an instrumented version of the device driver for the Interlan NI1010 Ethernet controller [Inte82], a set of programs providing a front-end interface to the instrumented driver and display and analysis tools. ILMON allows the user to observe and analyze the traffic on an Ethernet filtered on criteria which he defines. The features of ILMON will be compared to those of the Excelan Nutcracker [Exce85] and of the network performance tools available on Sun3 workstations. The efficiency of ILMON, in terms of packet loss versus throughput, will be discussed for various monitoring functions. The current implementation of ILMON runs on a DEC VAX<sup>‡</sup> 11/750 equipped with the previously mentioned Interlan controller.

## 1. INTRODUCTION

ILMON is a Local Area Network monitoring facility developed in conjunction with the Local Area Network Testbed (LANT) project [Moll83][Barn85] in the Department of Computer Sciences at the University of Texas at Austin. ILMON performs promiscuous reception of network traffic and presents the user with reports on the observed traffic in various formats. Packet reception may be conditioned on many quantities, including hardware source and destination addresses, packet length, packet type, and the error status of the packet. Programs also exist for consolidating data from several monitoring sessions and for formatting the data for graphical representation.

ILMON runs on a Digital Equipment Corporation VAX 11/750, which is connected to the main UT Campus Ethernet and an experimental network with Interlan NI1010 Unibus Ethernet Controllers. It uses the promiscuous reception and receive-on-error modes provided by the NI1010 to monitor network traffic. The 4.3BSD UNIX device driver for the NI1010 was augmented to provide various monitoring modes which the user configures by constructing a *filter* for the monitoring session. A filter identifies which of the available monitoring functions are to be active during a session and under what conditions a packet should be included in the collected information.

\* The work was performed while the author was in the Computer Science Department at the University of Texas at Austin.

† UNIX is a trademark of Bell Laboratories.

‡ DEC and VAX are trademarks of the Digital Equipment Corporation.

ILMON was originally conceived as the passive monitor for the Local Area Network Testbed. In that capacity it is used to collect information on the throughput and stability characteristics of a network running various experimental protocols. Other machines on the experimental network are designated as load generators and produce an artificial workload on the network during experiments. The monitoring software is, however, flexible enough to be of use in monitoring the performance of production networks for purposes of traffic and utilization analysis, and also has possible applications for network fault isolation.

The remainder of the report discusses at greater length the design, implementation, use, and possible extension of ILMON. Section two discusses the design of the system, including the method of configuring monitoring sessions using filters, the types of reports produced, and the practical limitations of the system. Section three discusses the implementation of the device driver extensions and the user interface for ILMON. Section four gives some preliminary results on the efficiency of ILMON and presents some utilization figures for the UT Campus Ethernet collected with ILMON. Section 5 discusses possible extensions to the system.

## **2. DESIGN**

The design of ILMON centers around the notion of using filters to specify the monitoring activities which will occur during a monitoring session. The user specifies a filter with a command interpreter which is loosely based on the UNIX `ifconfig` utility. The filter itself is a data structure containing flags indicating whether each of the possible monitoring activities is enabled for a session. There are also fields in the structure for various comparison quantities and storage for the data collected during a session. Most of the capabilities of the Excelsior Nutcracker are incorporated into this framework; the principal omissions are analogues to the Nutcracker capability to generate valid and erroneous packets on the network. All traffic generation in the LANT environment is done by the designated load generation machines.

### **2.1. The filter mechanism**

In general usage, the word *filter* suggests the desire to exclude or "filter out" something. Though ILMON also uses filters in accomplishing other tasks, the principal purpose of a filter is to exclude uninteresting packets from the data recorded during a monitoring session. Due to memory constraints and the large disparity between mass storage access times and the typical interval between network events, it is not practical to record all packets in their entirety for later examination and data extraction. Thus, it is necessary to perform some selection during the collection of data. In ILMON, filters provide a mechanism for specifying a predicate which a packet must satisfy before information concerning that packet is included in the collected data. The filter also determines which of several different data collection modes is in use during a monitoring session, and provides storage for the collected data.

### **2.2. Filter predicates**

ILMON allows packets to be filtered by any combination of several criteria. In the current implementation, to be included packets must satisfy the conjunction of all the criteria specified in the filter used for a session. The remainder of this section discusses each of the available criteria for packet inclusion in some detail.

#### **2.2.1. Error status**

The user may choose to include only packets received in error, only valid packets, or all packets. The Interlan NI1010 normally does not pass error packets to the device driver. However, it is possible to run the interface in "receive-on-error" mode, in which case error packets are not automatically discarded by the interface.

#### **2.2.2. Hardware addresses**

ILMON allows the user to filter packets on Ethernet source and destination addresses. To do so, a list of source addresses is specified and the source address of all incoming packets is checked against the list. If the source address does not match any of the addresses in the list, the packet is discarded. If a match is found, the packet is logged or summarized. A similar list is specified for destination addresses. If either list is empty, no checking on the corresponding address is done. This type of checking is useful for

monitoring the traffic between two stations, or monitoring the network output of a troublesome host.

### 2.2.3. Packet length

The user may specify a constant between the minimum and maximum allowable packet lengths and one of the relational operators  $>$ ,  $<$ , or  $=$ . Packets are included which are greater than, less than, or equal to the constant respectively. It is also possible to specify a range of lengths. In this case, packets are included in the totals only if their length falls within the range specified.

### 2.2.4. Packet type

Packets may be filtered according to the value in their header's type field. The possible types are taken from the most recent "Assigned Numbers." [Reyn85] Type information is often useful in characterizing the applications which contribute to network load.

## 2.3. Collected quantities

Once a packet has satisfied the filter predicate, the storage fields of the filter are updated with information about the packet. A number of bits in the filter flag are used to specify what sort of information is kept during a monitoring session. Certain quantities are kept for every session, regardless of the filter flag value. These quantities are:

- 1) the number of errors which occurred during the session
- 2) the number of collision fragments observed during the session
- 3) the number of packets lost during the session.

These quantities are taken from the NI1010's onboard statistics registers. The total reported for number of lost packets by the interface is actually a count of the number of times one or more packets were lost before the successful reception of a packet, so estimates of actual packet loss using this value are not completely accurate. In particular, any loss rate above 50% will be reported as 50% -- every packet received will have the "lost" flag set. The remaining types of data collection are described in the following sections. The packet loss rate for some of the data collection modes is discussed in section 4.

### 2.3.1. Packet logging

For short periods of time, it is possible to log the headers of packets as they arrive for later examination. Memory constraints limit the number of headers that can be retained at any given time to approximately 5000. This capacity allows roughly 15 seconds of header logging, given an average network utilization of around 1 Mbps. It is possible, from ILMON, to retrieve and store filters, thus extending the duration of a packet logging session, but performing this action causes some packets to be missed in the meantime. The information stored is the arrival time of the packet, the ethernet header, and the IP header, if the packet has type IP.

### 2.3.2. Address histograms

Histograms of the number of packets and bytes originating from or intended for each hardware address on the network may be collected. Since maintaining the data structures for such a histograms incurs a large amount of processing overhead, many packets may be lost while collecting data in this mode. Though not currently implemented, the capability of collecting the same type of histograms on IP addresses is planned.

### 2.3.3. Packet length histogram

A histogram on the size of observed packets may be collected. This information (the number of packets of each size) is also used to calculate the throughput for the network and total number of bytes observed in the monitoring session.

#### 2.3.4. Packet type histogram

A histogram on the types of the packets observed may be collected. The recognized types are taken from [Reyn85].

#### 2.4. Comparison to other monitors

Two other network monitoring tools are available at the University of Texas at Austin, the Excelan Nutcracker and the Traffic monitor utility from Sun. This section discusses the similarities and differences among these tools and ILMON.

##### 2.4.1. The Excelan Nutcracker

The Nutcracker is an 8086-based workstation with an enhanced network controller and a 20 megabyte hard disk drive. It is composed of several logical subsystems, each of which is responsible for some aspect of the network monitoring task, such as filtering the input stream or storing the desired portion of a packet to the disk. These subsystems are hierarchically arranged, and each has an *object* associated with it which specifies the behavior of a subsystem during an experiment. The user must build the objects for each subsystem interactively prior to running the experiment, in much the same way that filters are specified in ILMON. The specification, however, is at a lower level of abstraction for the Nutcracker.

The filtering performed by the Nutcracker is of a slightly different nature than that done by ILMON. The *filter subsystem* consists of twelve "receive channels," each of which can be monitored independently. Each channel has a filter object associated with it. Four of the channels handle error packets of various kinds. The filter objects for the remaining eight receive channels can be specified by the user. The filter object consists of an offset and a string of up to 128 *octet-unions*, each of which can be either a constant from 0 to FF (hex), a range of values, or "don't care" which matches all octets. The filter subsystem performs an octet by octet comparison of the pattern with each packet, beginning with the octet specified by the offset. Since the Nutcracker considers everything between the preamble and the CRC checksum to be data, this mechanism allows matching to be done on both header quantities and the data field of packets. This mechanism is more powerful than the filter mechanism of ILMON because it allows packets to be filtered on the contents of the data field. However, it is more difficult to specify experiments using the Nutcracker's method since the user must keep track of where the fields of interest are located within a packet and what data translation must be done in order to successfully match the desired quantities.

The data collected is very similar on both systems. Both systems allow packet logging or tracing. ILMON logs only the Ethernet and IP headers, while the Nutcracker allows an arbitrary "slice" of the packet to be logged, up to and including the entire packet. The Nutcracker also allows the examination of runt packets and collision fragments, which are automatically filtered by the NI1010 even in receive-on-error mode. The available statistics are almost identical. In this area, ILMON has one distinct advantage: extensibility. If the user is not satisfied with the statistics provided, the data files are available to him, and he can extend the analysis package to provide whatever statistics he is interested in.

##### 2.4.2. Monitoring tools for the Sun workstation

The software release from Sun Microsystems, Inc. accompanying their Sun3 workstations includes two network monitoring tools. The program *traffic* provides a graphical, real-time display of network behavior. The program *etherfind* allows the user to view the headers of packets whose contents have satisfied a logical expression.

###### Traffic

Traffic allows the specification of one or more filters which packets must satisfy before they are included in the display. Each filter checks one of the following conditions:

- source/destination of the packet, specified either by host or by network
- IP protocol family to which the packet belongs, e.g. TCP, UDP, ICMP, etc.
- length of packet within specified range.

*Traffic* allows more than one filter to be active at the same time. If multiple filters are defined, packets must satisfy the conjunction of all filters. The address checking is done on the IP address of the packet, not

the hardware address. As previously stated, IP address checking is not currently implemented in ILMON, nor is filtering on the IP type of packets.

*Traffic* displays information about the behavior of the network in a number of windows. Each window contains one of the following displays:

- packet size distribution
- usage of each IP protocol type
- traffic generated by each source (top eight displayed)
- traffic intended for each destination (top eight displayed)
- overall network utilization

Each display is a time varying histogram of either packets per second or percentage of total packets. The update interval is under user control for each window; the interval may be varied from 0.1 second to 10 seconds in 0.1 second increments. Larger intervals are not supported. The network utilization is displayed as a strip chart. If filters are defined, they apply to all windows. Other than dumping a snapshot of the screen, there is no mechanism for storing the information presented.

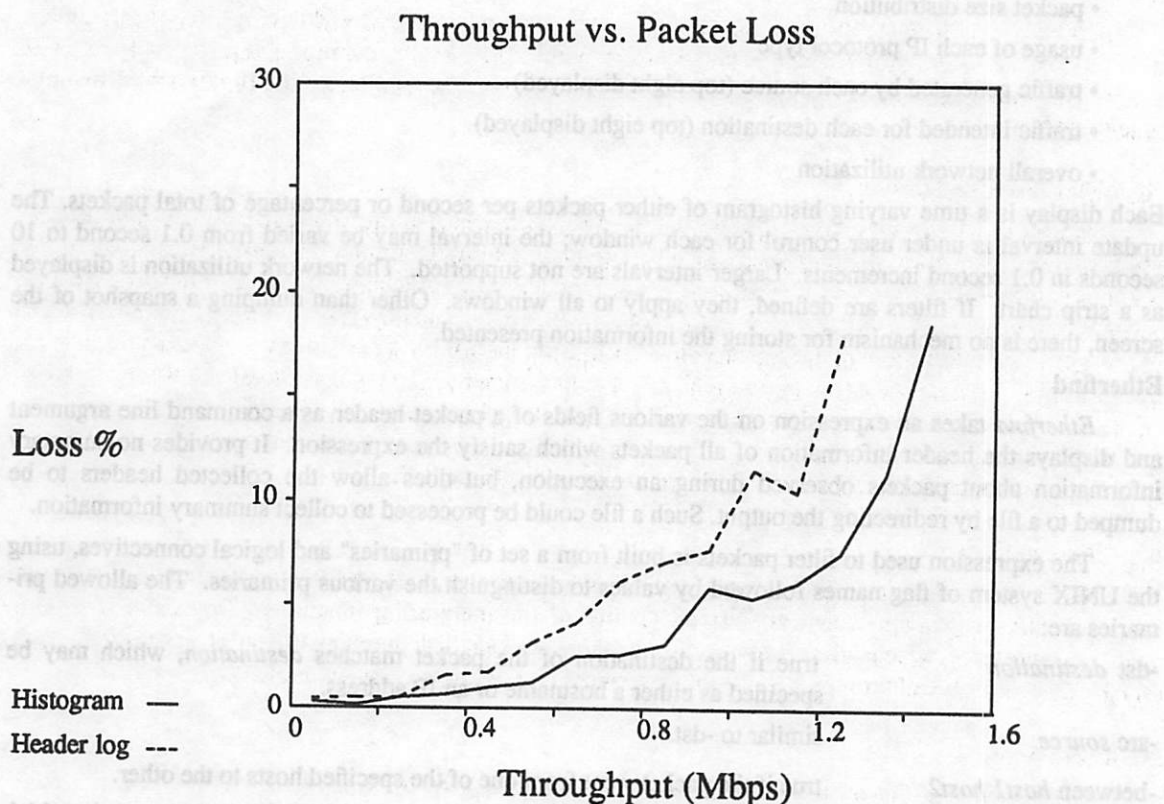
### Etherfind

*Etherfind* takes an expression on the various fields of a packet header as a command line argument and displays the header information of all packets which satisfy the expression. It provides no summary information about packets observed during an execution, but does allow the collected headers to be dumped to a file by redirecting the output. Such a file could be processed to collect summary information.

The expression used to filter packets is built from a set of "primaries" and logical connectives, using the UNIX system of flag names followed by values to distinguish the various primaries. The allowed primaries are:

-dst <i>destination</i>	true if the destination of the packet matches <i>destination</i> , which may be specified as either a hostname or an IP address.
-src <i>source</i>	similar to -dst.
-between <i>host1 host2</i>	true if the packet went from one of the specified hosts to the other.
-dstnet <i>destination</i>	true if the network part of the destination field matches <i>destination</i> , which may be specified as an address or a network name.
-srcnet <i>source</i>	similar to -dstnet.
-dstport <i>port</i>	true if the packet is of IP type UDP or TCP and has destination port value of <i>port</i> .
-srcport <i>port</i>	similar to -dstport.
-less <i>length</i>	true if the packet's length is less than or equal to <i>length</i> .
-greater <i>length</i>	true if the packet's length is greater than or equal to <i>length</i> .
-proto <i>protocol</i>	true if the packet is an IP packet of the type <i>protocol</i> .
-byte <i>byte op value</i>	true if byte number <i>byte</i> of the packet is in relation <i>op</i> to <i>value</i> . Legal operators are +, <, >, & and !.
-broadcast	true if the packet's destination address is the broadcast address.
-arp	true if the packet's type is ARP.
-rarp	true if the packet's type is RARP.
-ip	true if the packet's type is IP.

Primaries may be grouped using parentheses. Juxtaposed primaries are connected by a logical *and*. Primaries separated by '-o' are connected by a logical *or*. While this method of specifying expressions is fairly comprehensive, it is difficult and confusing for expressions of any complexity. For instance, checking for a particular hardware address would require six '-byte' primaries.



**Figure 1.** Efficiency of packet length histogram collection and packet header logging. Averaged from observations of the UT Campus network.

*Etherfind* parses and displays the header of each packet that satisfies the expression. IP source and destination addresses are automatically converted to hostnames. The header can optionally be dumped in hexadecimal.

## 2.5. Limitations

Along with the shortcomings noted in relation to other network monitoring systems, there are several practical limitations to ILMON's capabilities. ILMON is not able to log collision fragments due to the fact that the Interlan controller filters them out automatically. This feature of the controller is not programmable in the way that the filtering of error packets is. Being able to examine the fragments themselves is a desirable capability in fault diagnosis tasks. In addition to this hardware limitation, the efficiency of the ILMON software is less than perfect. The instrumentation of the controller device driver introduced extra code into the time-critical receive interrupt routine. This means that when monitoring is in progress, some packets are not processed before they are overwritten in the controller's buffers. A quantification of this packet loss appears in section 4.

There were also useful features which were considered but not included in the design of ILMON. Among these are real-time display of collected data and a more flexible scheme for specifying the filter predicate. ILMON is not written for a particular workstation or graphics device, and thus does not incorporate the visual display of collected data in real time as *traffic* does. In light of this fact, the further reduction of efficiency which would have been incurred by retrieving and displaying the data as it was collected

was deemed counterproductive. As previously stated, the various conditions specifiable in the filter are connected by logical *and*. This was convenient and sufficient for the author's purposes, but may be expanded in the future to incorporate grouping and disjunction.

### 3. IMPLEMENTATION

ILMON is implemented as an instrumented version of the 4.3BSD UNIX device driver for the Interlan NI1010 Ethernet controller and a front end which allows the various monitoring options to be selected by the user. There are also several programs for displaying the filters collected and for consolidating and reducing the data.

#### 3.1. Driver instrumentation

Monitor functions are activated using the UNIX ioctl mechanism to communicate between the user application and the device driver. Code was added to the *ilioc* routine in the Interlan driver to pass filters in and out of the kernel, to set and clear promiscuous mode, and to reset and retrieve the on-board statistics registers. Whenever the interface is in promiscuous mode (as indicated by a bit in the flag field of the *ifnet* structure for the interface) code in the receive interrupt routine *ilrint* is executed to test packets against the filter predicate. If a packet satisfies the predicate, the filter's counters are updated to reflect the reception of the packet, and the headers are stored if packet header logging is enabled. Finally, the address of the packet is checked. If the destination is the monitor station or the broadcast address, the packet is enqueued for the proper higher level protocol. Otherwise, the packet is discarded.

#### 3.2. User interface

The ILMON user interface is composed of several programs. ILMON is an interactive front end, loosely based on *ifconfig*, which allows full control of the monitoring functions. *Timedmon* is a non-interactive program which allows monitoring sessions of specified duration to run at a specified time. Finally, there are several programs for displaying and analyzing data collected by ILMON.

##### 3.2.1. ILMON

ILMON allows the user to specify, run, and save the results of monitoring sessions. The filter specification process is menu-driven, presenting the user with a list of the possible monitor functions and predicate conditions. Additionally, the user may define filter templates, which, once saved, can be reloaded and used in monitoring sessions at later times. These templates are also used in the non-interactive program *timedmon*. Several other utility functions are available in ILMON, such as retrieving the Interlan's onboard statistics, reading the interface flags from the *ifnet* structure, reading and setting the driver state flags from the *ilsoftc* structure, and reading the interface's control registers. Many of these functions are left over from an earlier program used to debug the added ioctl code in the device driver.

##### 3.2.2. Timedmon

ILMON can be viewed as needing constant supervision. *Timedmon* automates the monitoring process, taking all the information it needs to run a monitor session (or a series of monitor sessions) as command line arguments. *Timedmon* requires that a filter template have been previously defined and saved in ILMON. The syntax for *timedmon* is then

```
timedmon -iinterface # -ffilter -dduration -sstart time -ooutput file -rrepetitions
```

*Interface* is the unit number for the interface to be monitored. *Filter* is the name of a file containing a filter template defined in ILMON which will be used for the monitoring session. *Duration* is a string of the form *hh:mm:ss* specifying how many hours, minutes and seconds the monitoring session is to last. *Start time* is the time (in 24 hour notation) at which the monitoring session is to begin. *Output file* is a name to be used for saving the resulting data, or a prefix from which file names will be built should multiple sessions be requested. *Repetitions* is the number of times the monitoring task should be repeated. This program allows monitoring session requests to be set up ahead of time and left in the background to wait for their specified starting time. It also provides the ability to run sessions for a specified period of time, a function not provided in ILMON.

## Ethernet Utilization -- six minute samples

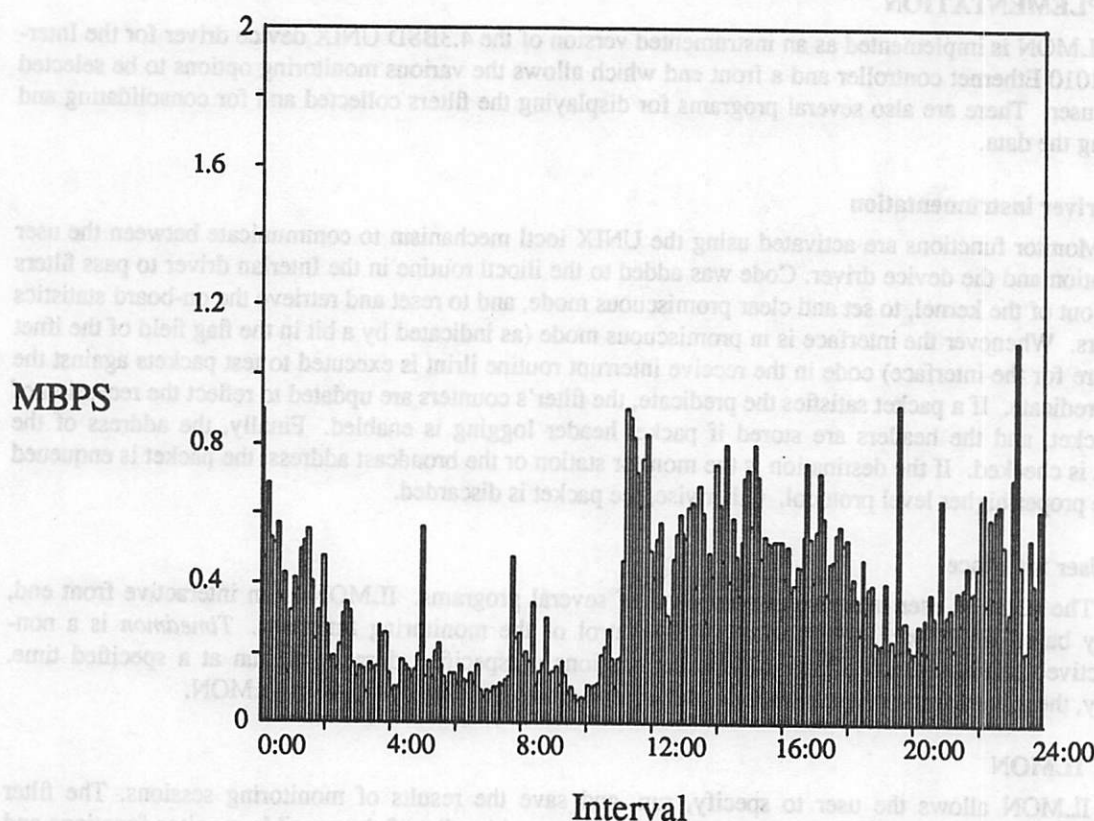


Figure 2. 24 Hour Ethernet utilization, six minute samples.

Table 1: Xerox vs. ILMON studies

Quantity	Shoch	ILMON
Packets/day (mil.)	2.2	8.7
Bytes/day (mil.)	300	3,800
Avg. Utilization	0.8%	3.6%
Peak Util. (6 min period)	7.9%	11%
Min. Util. (6 min period)	0.2%	0.7%
Mean packet size	122	439
Avg. inter-packet time (ms)	39.5	7

### 3.2.3. Data examination and processing

Data analysis functions are provided by the Experiment Analysis Package (EAP), a part of the Local Area Network Testbed software. EAP provides some frequently used plots as well as the ability to specify general plots on any field of the filter's data. The standard plots available are: packet length histogram, a raw plot of network utilization data, and an averaged plot of network utilization data. Definition of the axes and labeling is under user control. As work on the testbed proceeds, further plots and data treatments will be added. The general plotting facility allows the user to choose quantities derived from filter data

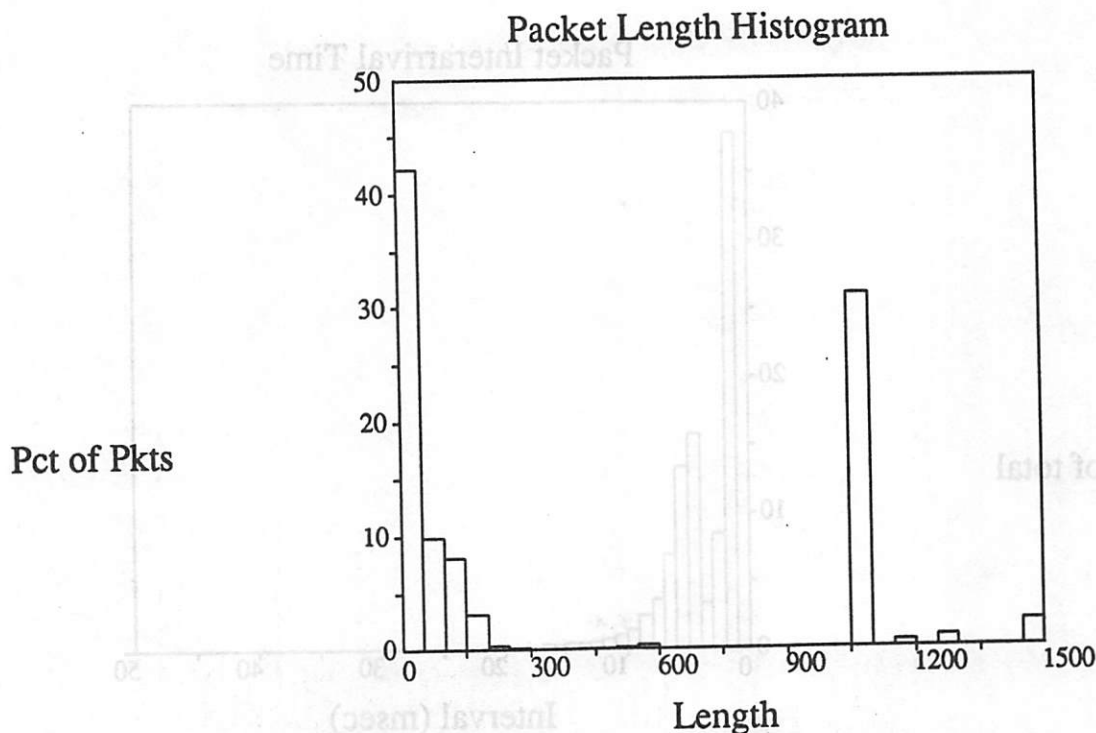


Figure 3: Packet length vs. percent of total packets.

such as throughput, utilization, packet size, packet loss, etc. for the axes of a plot. This facility is easily extensible. EAP works on single filters, or can consolidate the data from many filters for plotting. All of the figures for this report (except for figure 4) were generated using EAP.

There are two other programs which are useful for viewing and analyzing filters. *Prfilter* simply reads a filter and prints out the contents in human-readable form. The output of *prfilter* is suitable for piping through *lpr*, the UNIX print spooler. *Logsum* produces a summary of filters containing logged packet headers. The information produced includes the timestamps of the first and last packets logged, the number of packets logged, the throughput, the average inter-arrival interval for the logged packets, and a histogram of the inter-arrival time distribution. Figure 4 was produced by *logsum*.

#### 4. PERFORMANCE

The extra processing required in the network interface device driver for collecting data often causes the interface to miss some packets. The penalty thus incurred is investigated for packet histogram collection and header logging.

##### 4.1. Packet loss for various monitoring functions

Figure 1 shows the percentage of packets lost during the collection of packet length histograms and packet header logging. Packet length histogram collection is the minimal monitoring task performed by ILMON, requiring only that one array element be incremented, along with the overhead incurred by any of the monitor functions. This overhead consist of several flag comparisons and at most two Ethernet address comparisons to determine whether a packet should be discarded or passed on to the higher level protocols. On the other hand, header logging requires that the packet header be copied into a log structure, and may require additional copying depending on the packet type. The data plotted in figure 1 was averaged from one minute samples, with bins of 100,000 bits and, where possible, with 20 samples per bin. No users were logged in to the monitor computer while most of the samples were collected, though the computer was

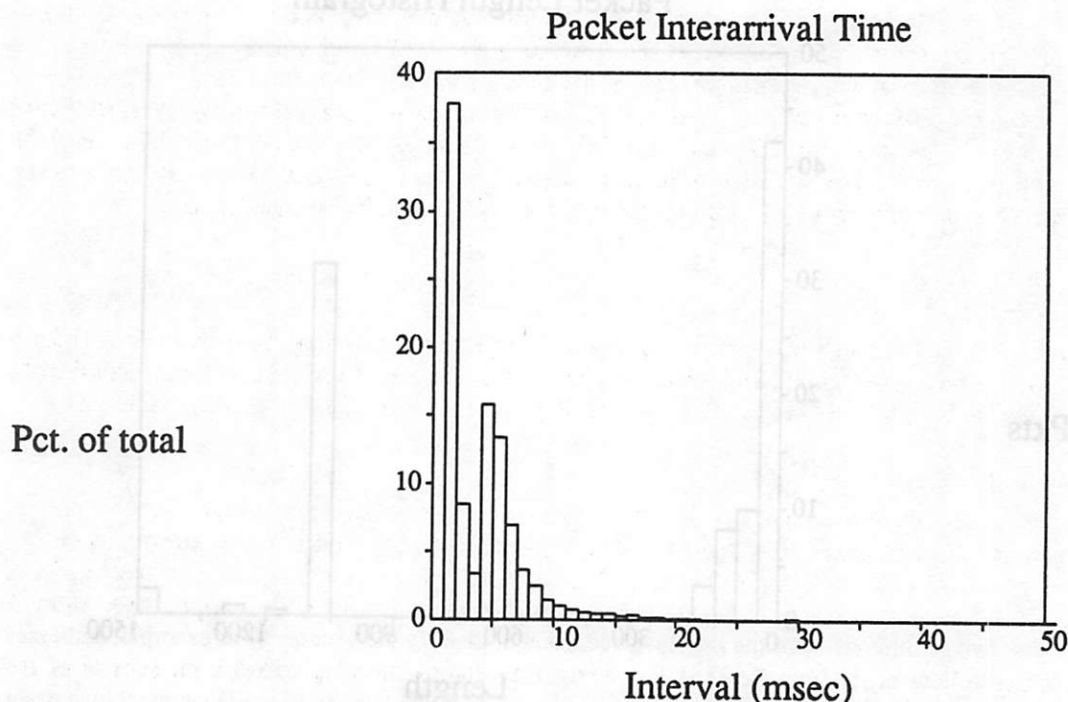


Figure 4. Inter-arrival time distribution. Taken from one thirty second packet logging sample.

operating in multi-user mode. Packet loss for histogram collection remains under 10% for throughput levels up to 1.2 Mbps. At higher throughput, more serious degradation occurs. However, since throughput levels greater than 1.2 Mbps for one minute intervals were rarely observed, complete sets of samples for these levels were not collected. Trials of LANT artificial load generation software indicate that the effective resolution of the Interlan's loss reporting mechanism (50% reported packet loss) is reached at throughput levels of approximately 3.0 Mbps. The loss percentage for header logging was, as expected, consistently greater than for histogram collection, exceeding 10% at a throughput of about 1.0 Mbps.

#### 4.2. Utilization of the UT Campus Ethernet

ILMON has been used to collect utilization data on the University of Texas campus Ethernet. The network connects some 80 nodes, including mainframes, minicomputers, workstations, file servers, laser printers and terminal concentrators. Many other machines reside on subnets and contribute to the network's traffic. Several higher level protocols, such as the DOD/ARPA protocol suite, DECnet, and CHAOSNET, are used.

The network was monitored for a 24 hour period, with samples summed over six minute intervals. During this period, 8.7 million packets were transmitted, containing 3.8 billion bytes of information, not including the Ethernet headers, preambles and frame check sequences. The average throughput was 0.358 Mbps, for a utilization of 3.58%. The peak utilization was 11.03%, and the minimum utilization was 0.7%. The results of this observation are shown in figure 2. Figure 3 is a histogram of observed packet lengths. This figure shows the classic bimodal distribution, with a large percentage of small packets carrying terminal traffic, and a smaller percentage of very large packet associated with file transfers. The second group is somewhat exaggerated in the UT environment by the presence of a cluster of diskless Sun workstations and their file server on the backbone network. The Sun Network Disk (ND) protocol has a block size of 1072 bytes. Most of the traffic with packet sizes larger than 1072 are generated by various applications using the User Datagram Protocol. [Post80] Figure 4 is a histogram of the observed interval between packet

receptions for the monitor node.

A comparison to a similar set of observations reported in [Shoc80] is shown in table 1. The Shoc data was collected on the Xerox PARC experimental Ethernet, which ran at a rate of 3 Mbps, while the UT Campus Ethernet is a 10 Mbps network. Though this is a less than perfect comparison, the average utilization, peak utilization, and minimum utilization figures are of interest. These quantities indicate that though the two networks were of roughly similar size at the time of the measurements, the UT network was significantly busier. Though the small average packet size for the Xerox experiments is partly an artifact of the PUP protocol in use there, the difference also indicates that subsequent protocol designs have used the medium more efficiently. The observed differences between the two networks also reflect the growing dependence of a broad spectrum of computer tasks on Local Area Network communication.

## 5. EXTENSIONS

Though ILMON is a useful and powerful tool in its present form, several possible extensions have been identified. These extensions fall into three categories: more powerful address checking, more flexible filter predicate specification, and further development of the data reduction and presentation facilities.

### 5.1. Address checking

As previously noted, address checking will eventually be expanded to include checking IP addresses as well as hardware addresses. Currently, the hardware address checking works only on complete Ethernet addresses; that is, all six bytes of the address must be specified. Ethernet addresses are assigned by Xerox such that all interfaces manufactured by a vendor have addresses which fall within some range. In most cases, the first two or three bytes of the interface address will identify the vendor. (For example, addresses of Interlan controllers begin with 02.07...) Extending the address checking to work on subsets of the hardware address would allow the user to take advantage of such knowledge to check on machines using the same type of network interface hardware. In the case of IP addressing, the benefits are even greater for this type of address checking; checking only a portion of the IP address (as *traffic* does) would allow the user to isolate traffic from specific subnets connected to a backbone network.

### 5.2. Filter predicate specification

The flexibility of ILMON's filtering method would be greatly improved by allowing the user to specify the grouping and logical connectives for multiple filter conditions. In the current application of ILMON, the use of the conjunction of all filter conditions is adequate; however, this extension would greatly enhance ILMON's value as a general network monitoring tool. In addition, the type checking will be extended to break down IP packets by the protocol they belong to, e.g. TCP, UDP, etc.

### 5.3. Data reduction and presentation

The current data reduction and presentation facilities consist of a program which collects the data from multiple filters and produces reports of averages and data files for graphs. The graphing facility allows any of several reduced quantities (throughput, packet rate, loss rate, etc.) to appear on either axis and produces a data file for a plotting package that generates plots in *pic* [Kem82] format. This is very convenient for including graphs in documents. Currently, the reduction program performs only simple smoothing on data for plotting. The addition of various types of curve fitting and extrapolation is planned.

## 6. SUMMARY

This paper has presented a description of ILMON, a tool for configuring and running network performance monitoring sessions, and several utility programs connected with the use of ILMON. ILMON allows a user to collect information about network traffic which satisfies a filter predicate on the form and contents of the packet. Several programs for the display and analysis of data thus collected are provided. Comparisons to the Excelan Nutcracker and the Sun3 *traffic* and *etherfind* utilities were given. The full implementation of ILMON performs well for network loads up to approximately 1.2 Mbps. At higher loads, packet loss due to monitoring overhead reaches unacceptable levels. For most normal network monitoring activity, this performance is sufficient. However, for use in the Local Area Network Testbed

environment, where protocol behavior at extreme loads is of interest, further optimization will be necessary. This optimization will consist of paring down the code in the Interlan device driver to provide only those functions necessary to performance studies. Features such as address checking, packet type histograms, etc. will be eliminated.

## ACKNOWLEDGEMENTS

This work was partially supported by National Science Foundation grant MCS8122039.

## REFERENCES

- Barn85 B. Lewis Barnett and Michael K. Molloy, "Local Area Network Testbed Design," Technical Report TR85-25, Department of Computer Sciences, University of Texas at Austin, May 1985.
- Exce85 *Nutcracker User Manual*, Excelan, Inc., San Jose, California, 1985.
- Inte82 *NI1010 UNIBUS Ethernet Communications Controller User Manual*, Interlan, Inc., Chelmsford, Massachusetts, 1982.
- Kern82 B. W. Kernighan, *PIC - A Graphics Language for Typesetting*, revised edition, March 1982. (online UNIX documentation)
- Moll83 Michael K. Molloy, "Experimental Evaluation of New CSMA Protocols," *Proceedings of the National Communications Forum*, October 24 - 26, 1983, pp. 350 - 354.
- Post80 J. Postel, "User Datagram Protocol," RFC 768, Information Sciences Institute, August 1980.
- Reyn85 J. Reynolds and J. Postel, "Assigned Numbers," RFC 943, Information Sciences Institute, April 1985.

## Virtual Disks: A New Approach to Disk Configuration

Thomas Van Baak  
Pyramid Technology Corporation  
1295 Charleston Road  
Mountain View, CA 94039-7295  
pyramid!tvb

### ABSTRACT

The traditional UNIX technique of disk partitioning and file system configuration has changed very little over the years. A growing frustration with the limited flexibility of the traditional scheme led to our development of the virtual disk. A virtual disk pseudodevice driver has been implemented which accomplishes the following:

- Disk partitions and file systems can be created up to 4 gigabytes in size through disk partition concatenation.
- Disk file system performance is dramatically increased through disk partition striping.
- Data integrity is enhanced through disk partition mirroring.
- Memory disks can be concatenated with physical disk partitions to create virtual disks with sub-millisecond access times which are not limited in size by the amount of physical memory.

All these features are cleanly implemented by the virtual disk facility in a way completely transparent at both the user-level and the disk device driver level.

This paper describes the new virtual disk facility and its machine independent implementation. In addition, the results of a large number of experiments in disk performance are presented which demonstrate both the flexibility and the performance of virtual disks.

### 1. Description of a Virtual Disk

A virtual disk is a device that functions identically to a traditional physical disk, but whose relation to physical disks is determined from a mapping of a physical disk (or disks) to a virtual disk. This is done by means of a virtual disk configuration file, `/etc/dktab`. The administrative utility `/etc/dkconfig` uses `/etc/dktab` to perform the mapping. The administration of virtual disks by `dkconfig` and `/etc/dktab` file resembles the way `mount` and `umount` administer the mounted file systems according to the specifications given in `/etc/fstab`.

From the perspective of a user program, the interface to a virtual disk partition is indistinguishable from that of a physical disk. Both the block and the character device disk interfaces are supported.

The virtual disk facility does not change the existing disk partitioning scheme in any way; both the existing physical disk partition facility and the virtual disk facility may be used together without incompatibility.

Virtual disks provide a level of disk access not directly tied to a particular disk drive, controller, or I/O controller subsystem. A virtual disk may span several disk drives (of any type) and/or several controllers (of any type).

There are four types of virtual disks, as follows:

1. A *simple* virtual disk consists of a single piece of a single physical disk drive. In this respect, a simple virtual disk is identical to the notion of a disk partition. An offset and length may be optionally specified that allows one to further subdivide an existing physical disk partition. If the *c* partition is used (which is conventionally the entire user-accessible portion of a single disk), one can define new and arbitrary partitions independent of the hard-coded partition tables configured in the kernel. This allows soft disk partitioning, without restrictions to names or number of partitions. It can be used, for example, to create many small partitions on a single disk or within a single disk partition.

This implies that all disk partitions can be created using virtual disks. The only reference to a physical disk in the kernel-based partition tables would be for the *c* partition. All further partitioning would be defined with virtual disks in */etc/dktab*.

2. A *concatenated* virtual disk consists of two or more pieces of one or more physical disk drives. It is the logical sum of two or more disk partitions. This is the reverse of the notion of partitioning a disk into smaller sections. Instead, one can combine disks into one large logical disk.

The concatenated disk permits logical disks or file systems up to 4 giga-bytes in size (the limit of a 32-bit seek address). All I/O to a virtual disk is identical to I/O on a physical disk of a capacity equal to the sum of the pieces of the virtual disk. The physical pieces of a concatenated virtual disk may be of any size and in any order.

3. A *striped* virtual disk, like concatenated virtual disk, also consists of two or more pieces of one or more physical disks. However, instead of the composition being the logical sum of each piece, an interleave algorithm is employed in translating block numbers of the virtual disk into those of the physical disk. This allows sequential I/O on the virtual disk to be translated into interleaved I/O across two or more physical disks. The cluster size of a striped partition determines how the blocks of sequential I/O are distributed among the two or more pieces of the physical disks. The cluster size may be as small as one sector or as large as the size of the physical disk itself. In the latter boundary case, the striped virtual disk is identical to a concatenated virtual disk.
4. A *mirrored* virtual disk is a specification of two or more disks (either physical or virtual) which appears to the user as a single disk. However, all writes to a mirrored virtual disk are translated into a write to each piece of the mirrored disk. And any reads to a mirrored virtual disk are translated into a single read from any one of the pieces of the virtual disk.

If an I/O error occurs on any piece of the mirrored virtual disk, that piece is deactivated, but mirrored I/O will continue on all other pieces and the I/O error is hidden from the user. Only when every piece of a mirrored virtual disk has been deactivated will the user receive the error.

# Disk Response Time Measurements

Thomas D. Johnson  
Jonathan M. Smith†  
Eric S. Wilson

Bell Communications Research  
3 Corporate Place, Piscataway, NJ 08854

## ABSTRACT

This paper describes response time measurements made by inserting tracing code in the UNIX® System device driver for the Digital Equipment Corporation™ (DEC®) RA81 disks attached to a DEC UDA50 controller.

The goals and methodology of the measurements are described. Detailed analysis of data derived from the measurements is provided, and some conclusions are drawn about the effect of UNIX file system mechanisms on response times as seen by user processes.

DEC RA81 Disk Drive Characteristics	
Cylinders	5216
Transfer Rate	2.2 megabytes/second
Rotational Speed	3600 rpm
Average Rotational Latency	8.33 milliseconds
Head Switch Latency	6 milliseconds
Average Seek	28 milliseconds
One Cylinder Seek	7 milliseconds
Maximum Seek	50 milliseconds

Up to four RA81 drives can be connected to the UDA50 controller; the UDA50 is a UNIBUS device. The DEC UNIBUS subsystem interfaces to the synchronous bus interface (SBI) through a UNIBUS Adapter (UBA). The UBA can support a maximum data transfer rate of only 1.33 megabytes/second. Thus, the peak data transfer rate of the drives is not realizable with this configuration.

This section describes system and performance measurement software available to our study.

1.2.1 Operating System  
Our VAX systems operate UNIX System V Release 2. Virtual memory is achieved via swapping; paging is not used in this Release. As supplied by AT&T Bell Laboratories, UNIX does not have a device driver for the UDA50/RA81. Our configuration was originally operational using a device driver adapted from a 4.3BSD driver for the UDA50; the driver's derivation from 4.3BSD led to the omission of features for reporting performance via the SAR(0) system. SAR disk performance reporting was added by a few changes in the driver.

† Jonathan M. Smith is currently at Columbia University, on leave from Bell Communications Research, Inc.

\* UNIX is a registered trademark of AT&T.

\*\* VAX, Digital, UNIBUS, MASSBUS, SBI, UDA50, RP06, HSC50, VAXCluster, RA81, DSA, and DEC are Trademarks of Digital Equipment Corporation.

# Disk Response Time Measurements

Thomas D. Johnson

Jonathan M. Smith†

Eric S. Wilson

Bell Communications Research  
3 Corporate Place, Piscataway, NJ 08854

## 1. Introduction

This report describes measurements made in the course of a disk performance study. These measurements deal with the response time for access requests made to the DEC RA81 disk, as measured on a system used for general-purpose time-shared computing. This differs from some previous work<sup>[1] [2]</sup>, much of which is directed towards measuring and improving file system throughput. The paper assumes a knowledge of UNIX System V internals as discussed by Bach<sup>[3]</sup>.

### 1.1 Equipment

The study is of a specific machine configuration, which consists of a DEC VAX-11/785 with 8 megabytes of main memory; mass storage devices are attached to a DEC UDA50 controller<sup>[4]</sup>. The associated disk hardware is the DEC RA81 disk drive; typical systems have four drives. The RA81 is a random-access, moving-head disk drive with non-removable media. The drive has a data storage capacity of 456 megabytes. The performance of the RA81 hardware is encapsulated in the following table, derived from the RA81 Disk Drive User Guide<sup>[5]</sup>:

DEC RA81 Disk Drive Characteristics	
Cylinders	2516
Transfer Rate	2.2 megabytes/second
Rotational Speed	3600 rpm
Average Rotational Latency	8.33 milliseconds
Head Switch Latency	6 milliseconds
Average Seek	28 milliseconds
One Cylinder Seek	7 milliseconds
Maximum Seek	50 milliseconds

Up to four RA81 drives can be connected to the UDA50 controller; the UDA50 is a UNIBUS device. The DEC UNIBUS subsystem interfaces to the Synchronous Bus Interconnect (SBI) through a UNIBUS Adapter (UBA). The UBA can support a maximum data transfer rate of only 1.35 megabytes/second<sup>[6]</sup>. Thus, the peak data transfer rate of the drives is not realizable with this configuration.

### 1.2 Software

This section describes system and performance measurement software available previous to our study.

#### 1.2.1 Operating System

Our VAX Systems operate UNIX System V Release 2. Virtual memory is achieved via *swapping*; *paging* is not used in this Release. As supplied by AT&T Bell Laboratories, UNIX does not have a device driver for the UDA50/RA81. Our configuration was originally operational using a device driver adapted from a 4.2BSD driver for the UDA50; the driver's derivation from 4.2BSD led to the omission of features for reporting performance via the SAR<sup>[7] [8]</sup> system. SAR disk performance reporting was added by a few changes in the driver.

Each of our systems are configured with 1000 buffers.

† Jonathan M. Smith is currently at Columbia University, on leave from Bell Communications Research, Inc.

### 1.2.2 Measurement Tools

The measurement tools which existed previous to this study consisted mainly of the various pieces of the SAR package. The information provided by SAR was inadequate for the purposes of determining the causes of poor response times from our disks. Porting the disk activity reporting portion of SAR would require extensive driver updates, and we felt that some better measurements were necessary.

### 1.3 Tracing: Initial Configuration

Several goals were clear at the outset:

1. We needed a way to measure *actual* disk activity, rather than logical reads and writes.
2. The data gathering should not contaminate the measurements; that is, it should not cause extraneous data to be generated<sup>1</sup>.
3. While it may add a few instructions to the service routines, the data gathering technique should not significantly alter the performance of the subsystem; otherwise timing data are meaningless.
4. It should give us enough auxiliary information for further detailed analysis with other tools.
5. The observations should be independent of the driver software.

Our approach was to implement a circular trace buffer of disk requests. It was used to track requests made to the driver through the *udstrategy()* routine<sup>2</sup>. The data gathered are a set from which we believe much can be derived; they are packed into a pair of 32 bit words; the inclusion of each datum is justified as follows:

1. R/W flag - High order bit of the first record. This was originally not included, but after preliminary data gathering was complete, it was clear that the direction of the I/O was important (e.g. for determining swapping characteristics)
2. Device number - 15 bits of a 16 bit quantity. In practice, all we need here is the 8 bit *minor* device number<sup>3</sup>, plus a bit to distinguish between character special and block special accesses; but it is easier the way we do it.
3. Timestamp - implemented by using the last 16 bits of the *lbolt* software clock. *Lbolt* is a long integer which is incremented upon each clock interrupt ("tick"); it is incremented at 1/60 second intervals. This clock value provides the smallest granularity with which we can measure events in software<sup>4</sup>. Thus we can determine the rate at which requests are queued, and the intervals between events such as bursts of writing. 16 bits seem sufficient; we could give up a few in the device number field if more seem useful.
4. Size of the request - this is not meaningful for buffered I/O, since all requests will use the file system block size. It is useful in determining the amount of I/O performed through the character special interface. We used 8 bits because it is all we had to spare in the second 32-bit word; but since *MAXBLK*<sup>5</sup> is 125 on our systems, it is not a problem. Requests larger than *MAXBLK* are the result of abnormal conditions (such as a system administrator using a huge blocking factor with *dd(1)*) and are therefore not relevant.

1. E.g., by using buffers which could otherwise be allocated.

2. This routine is called to place a buffer header onto the drive queue for the device. Once a buffer header is placed on the drive queue, the associated access will be performed. Thus, placing the tracing code in *udstrategy()* ensures that all disk accesses will be detected.

3. Indicating an instance of the device type determined by the first 8 bits, the *major* device number.

4. We could, of course, have modified the system's clock routines in order to achieve a finer granularity of measurement, but this would have required many modifications of the kernel.

5. The maximum size request made by the swapper in moving processes to and from the disk device.

5. The block number - we used 3 bytes (24 bits) because that's sufficient to store UNIX file system disk addresses. This block number is the block number relative to the beginning of the file system; the block number on the device can be determined off line by using information kept in the `ra_slices[]` structure defined in `sys/io.h`.

### 1.3.1 Buffer Implementation

The information described above requires two 32-bit words; we allocated a 2048 entry integer array, giving us 1024 records. Given the observed transfer rates of the block I/O subsystem, this was sufficient to capture all of the data. In addition, an integer was used to track the current index of the array.

### 1.3.2 Gathering

We gathered the records using the interface to the kernel's address space provided by `/dev/kmem`. The addresses of the trace buffer and the current index were obtained using the `nlist()` library routine. Data gathering consisted of monitoring the index variable for any changes; any records between changed values were dumped to the standard output. This output was redirected to the character special file interface for a magnetic tape device in order to minimize interference with the file system.

### 1.3.3 Formatting

The records from the tape device were formatted in a manner similar to the entries in `/etc/passwd`; that is, as colon-delimited strings, one line per record. This facilitated the use of tools such as `grep` and `sed` in analyzing the data. For example, records destined for the first slice of the root device via the raw interface could be selected using `grep`.

### 1.4 Tracing: Final Configuration

The remainder of this paper discusses the results derived from adding further tracing code to the UDA50 driver; this code traces both the *requests* made to the drive and the *responses* received from it. The methodology is discussed further in the section entitled "Data Gathering". We can analyze the results of this tracing to determine the behavior of the drive under UNIX file system activity.

## 2. Data Gathering

This section takes a "bottom-up" approach in describing how the data was gathered. By "bottom-up", we mean that the data gathering is discussed beginning at the lowest level (device driver) and ending at a discussion of the statistical analysis performed on formatted and processed data.

### 2.1 Tracing

In addition to the tracing described above, a circular trace buffer was added to the `udrsp()` routine, which handles responses from the drive<sup>6</sup>. We did not attempt to correlate the responses with the requests at this point as we anticipated that this would be costly in terms of driver performance<sup>7</sup>.

### 2.2 Gathering Trace Data

Gathering the trace data required a slight modification of the data gathering programs. The modification consisted of checking both request and response circular traces for new data and writing out any that were found. One problem with this unsophisticated approach is that the data generated does not retain its ordering with respect to time<sup>8</sup>; this has to be solved later when resolving the times to generate response

6. When the drive has completed an operation, it signals the CPU via the Unibus Adapter (UBA). The UBA is located at a known interrupt vector; when an interrupt occurs causing a processor trap, the `_Xuba` routine handles the trap by passing control to the `udintr()` interrupt handler for the UDA50. This routine examines the message header associated with the interrupt, and if it is a response, passes control to the `udrsp()` routine. Note that this processing is occurring at an extremely high priority level, and thus must be rapid.

7. On drives such as the RP06, correlation is trivial, as only one request can be outstanding, while we can have up to 14 requests outstanding on the RA81. The correlation would also be trivial if we used buffer headers to contain our trace data, but this approach would be undesirable for several reasons.

8. That is, a response might show up in the data stream ahead of the request which caused it.

statistics. A new flag was added to the output in order to indicate whether the record was a "Send" to the device, or a "Receive" from the device. The modified program determines this by the trace buffer it is examining, and outputs an integer flag which the formatting program uses to generate the appropriate record. The records are formatted:

send/receive flag:(major,minor):size:r/w flag:block number:timestamp

The sizes are in units of 512 byte blocks; the time stamp is provided by the Ibolt system clock, mod 65536. Some sample output (it is actual gathered data, but abridged for the sake of presentation) follows:

```
S: (32,0x10):2:R:1984:22127
S: (32,0x10):2:R:1992:22128
S: (32,0x6):2:R:486502:22129
S: (32,0x11):2:R:62782:22129
R: (32,0x10):2:R:1976:22123
R: (32,0x10):2:W:2152:22124
R: (32,0x10):2:W:2144:22124
R: (32,0x10):2:R:1984:22128
R: (32,0x10):2:R:1992:22130
R: (32,0x11):2:R:62782:22131
R: (32,0x6):2:R:486502:22132
R: (32,0x6):2:R:337822:22157
R: (32,0x6):2:R:339738:22158
S: (32,0x6):2:R:337822:22156
S: (32,0x6):2:R:339738:22157
```

This output can be used to illustrate several of the problems one encounters when trying to generate response times from the raw data, as discussed in the following section.

### 2.3 Generating Response Times

It turns out to be fairly easy to process such data visually: find a record preceded by an 'S', for a "Send", and then find the matching 'R', or "Receive", record. The difference between their timestamps gives the response time for the request. For example, the first line of the sample output is an 'S' record, for block number 1984. The corresponding 'R' is on the eighth line, and we compute the response time as (22128-22127) ticks, where a tick is 1/60 second.

There are, however, some potential problems in the data stream. For example, there are no 'S' records corresponding to the first two 'R' records, and the 'R' record for block 337822 on device (32,0x6) precedes the 'S' record. It turns out that some heuristics are necessary to get around the corrupted time-ordering of the data stream; these heuristics were incorporated into a program to analyze the data and produce response time figures on a per-request basis.

### 2.4 Response Time Statistics

There are several output formats available to the user of the response time analysis program. One is used to generate data for analysis with the S system<sup>[9]</sup>. Raw data were collected to obtain counts which could be used in later analysis. The first few lines of such an analysis are:

```
7480 responses took 0 ticks.
47922 responses took 1 ticks.
35755 responses took 2 ticks.
17120 responses took 3 ticks.
7821 responses took 4 ticks.
4734 responses took 5 ticks.
2803 responses took 6 ticks.
1667 responses took 7 ticks.
1203 responses took 8 ticks.
833 responses took 9 ticks.
```

The formatting of the data helped us in this task as it had in earlier analyses, because UNIX tools such as `grep` and `sed` could be used to aid the analysis. For example, inserting such a filter into the pipeline containing the response time analyzer allows us to analyze reads and writes separately.

### 3. Data Analysis

The amount of data gathered was tremendous; we monitored a system in our computer center for 3 hours on an afternoon in July, 1986. The gathered records took up about 6 megabytes of space; at 12 bytes per record, this was about 1/2 million records. It is clear that automated analysis of the data is not only preferable, it is necessary. Further, single statistics such as the mean and median<sup>[10]</sup> do not carry enough information to be useful in any non-trivial analysis of the system.

Graphical techniques, however, provide us with a methodology which is both compact and sufficiently informative. Once a subset of our data<sup>9</sup> was entered into S, we began the analysis.

#### 3.1 Simple Counts

The first plotted data, presented in Figure 1, is the number of responses which took a specified number of ticks.

9. Unfortunately, due to memory limitations, we are forced to use only the first 2000 of our gathered records to perform analysis other than simple counting. This sample had no responses taking longer than 200 ticks, unlike the larger sample examined for counting statistics. Examination of more data leads us to the conclusion that our sample is representative, and thus our conclusions are statistically valid.

Number of Responses vs. Response time

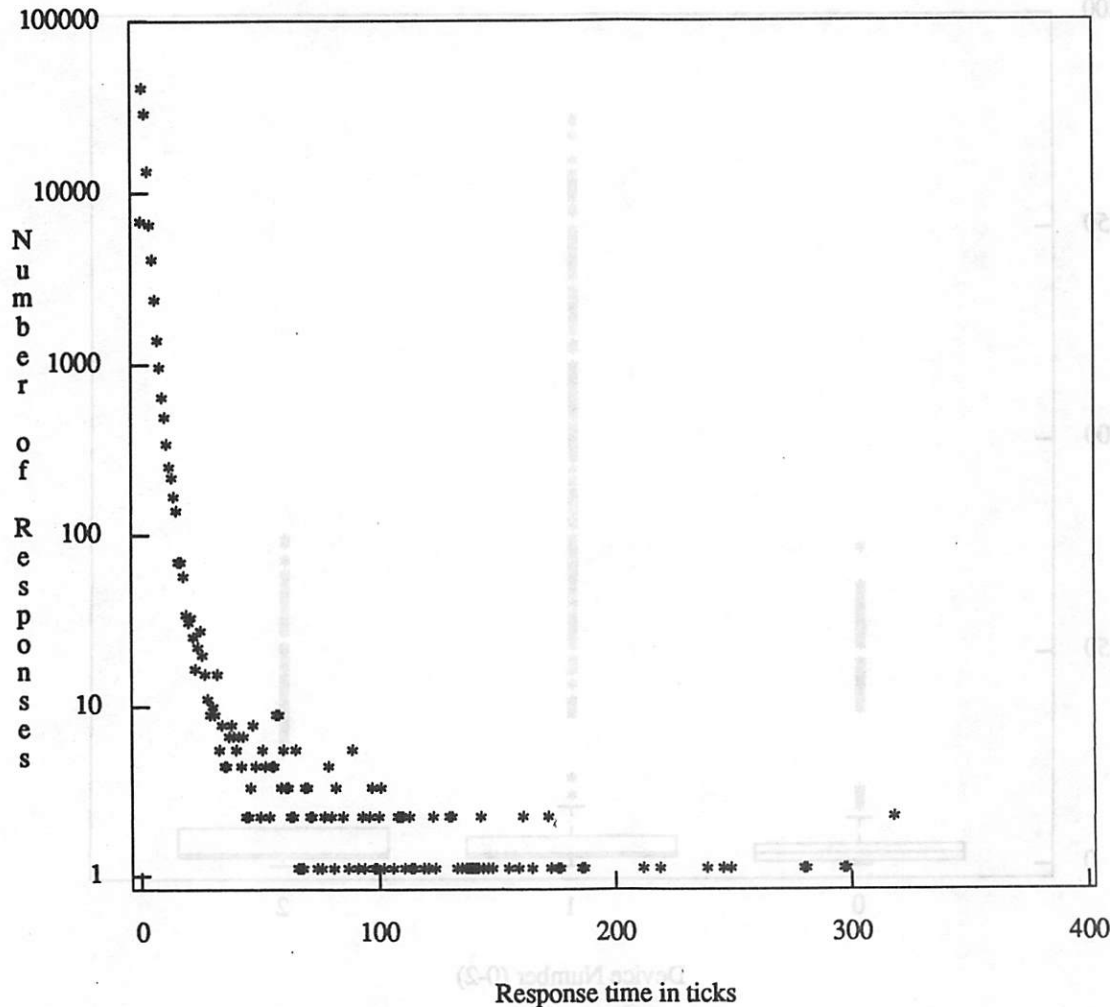


Figure 1. # Responses versus Response Time

We can see from this data that most of the responses were received in 5 or less ticks (note that the y axis is logarithmic). However, there are outliers in the greater than 300 tick range, indicating that *some* requests are seeing very bad service. While this won't affect the response time of a process performing writes<sup>10</sup>, reads will be affected, as the process cannot continue until the data is delivered. For example, if one of the blocks which required 300 ticks was in the middle of an a.out file being executed by some user, that command, however trivial, would require at least five seconds to begin executing.

### 3.2 Response By Drive

When performing disk load balancing, it is useful to know which drives are more heavily or lightly used. Thus information about the response time as a function of the drive may tell you which drives are candidates for exchanging some file systems.

10. UNIX writes are asynchronous; the write call returns as soon as the data is transferred into a system buffer. The contents of this system buffer will be written at some later time, either when the system needs the buffer or when a request to sync the disks is issued.

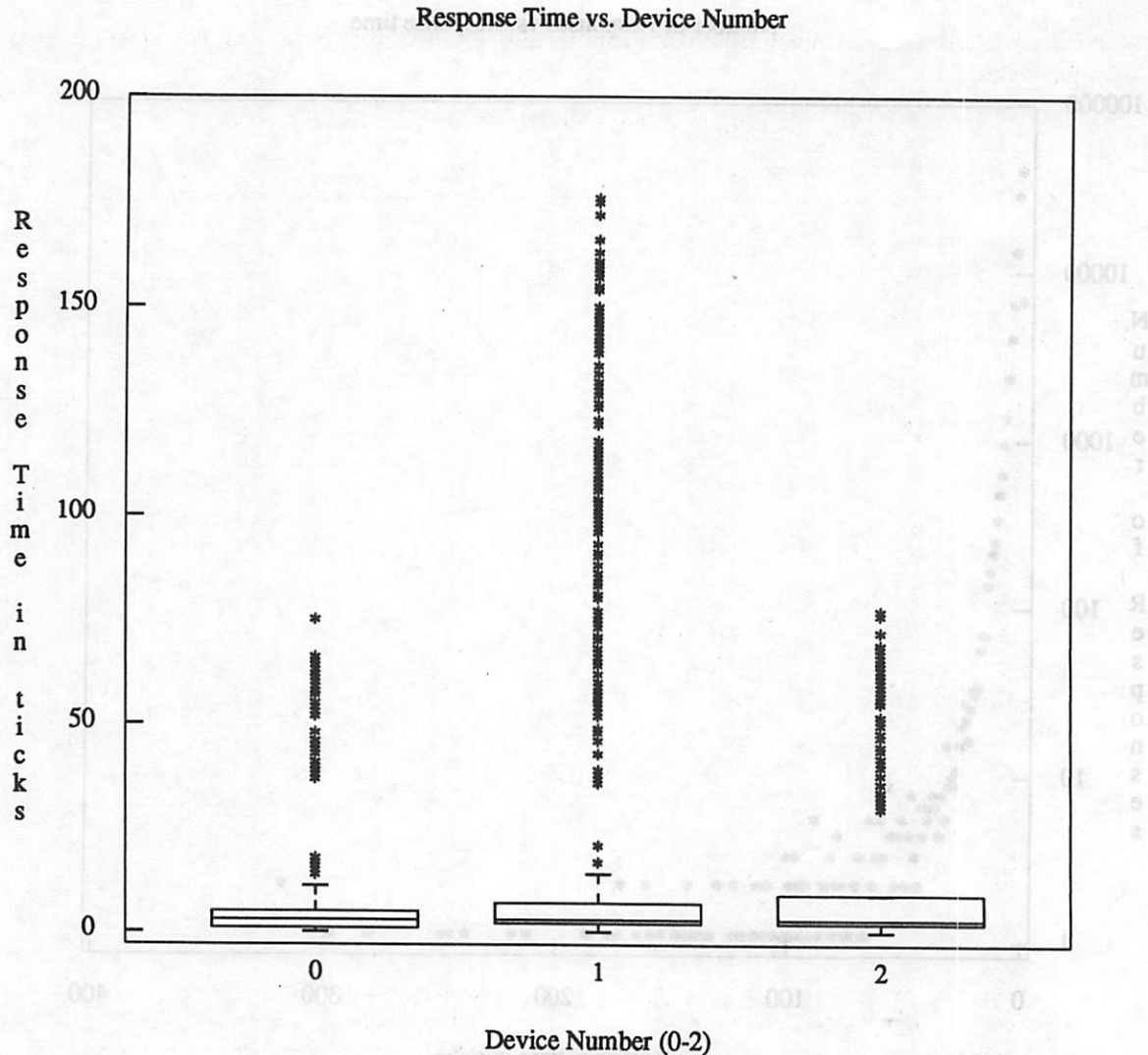


Figure 2. Response Time versus Device Number (0-2)

This graph<sup>11</sup> shows that although the worst response times are experienced by requests on drive #1, the medians of drive #0, drive #1, and drive #2 are fairly evenly matched. This evenness of the medians indicates that for this kind of file system activity, the loads seen by the drives is "balanced", indicating that the busy file systems are apportioned well between the three drives. Note, however, that the somewhat longer response times for the top 50 percent of requests on drive #2 may indicate some slowness; this information is lost if only the median is examined. If a particular drive or drives show unusual inequities in response times, it may prove useful to determine which slice(s) of the drive(s) are experiencing large numbers of accesses, and perhaps move some of that activity to another drive.

11. This particular presentation is called a *boxplot*; it provides much more information than an X-Y plot for this type of data. For a given x value, the box defines the middle 50 percent of the data, the horizontal line inside the box is the median, and the bar at the end of the dashed line marks the nearest value not beyond some standard range (in this case,  $1.5 \times (\text{inter-quartile range})$ ) from the quartiles. Points outside these ranges are shown individually. Details of *boxplot* presentation can be found in Chambers, Cleveland, Kleiner, and Tukey<sup>[11]</sup>.

### 3.3 Response by Block Number

Response by block number can tell us what areas of the disk are "hot spots" - places where there is a great deal of activity.

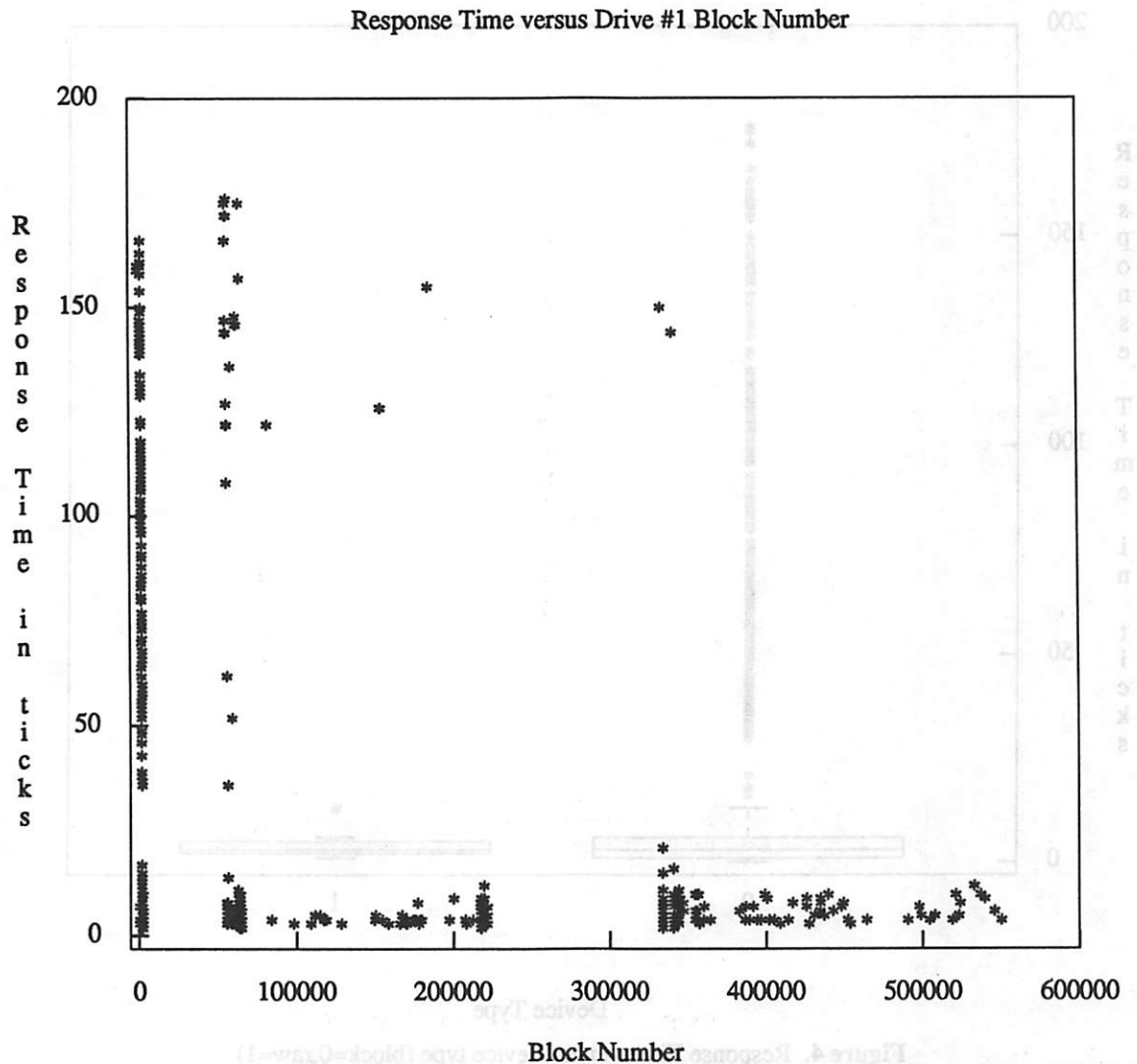


Figure 3. Response Time versus Block Number

For example, the above graph shows that certain areas of drive #1 experience very long response times, and that these areas are well-defined<sup>12</sup>. Some examination led to the deduction that these spikes are located at the *i-lists* of the file systems on drive #1. The spikes in response time result from a "sync" which causes many of these requests to occur in a large cluster.

### 3.4 Device Type

The character special or "raw" interface provides a means by which transfers of an arbitrary size can be accomplished; for example, swap I/O. The response times seen by raw I/O requests can tell you whether the disk transfer rate<sup>13</sup> or the time spent in the drive queue is the major source of delay experienced by

12. The size of a slice is 55692 blocks.

13. Although more about the disk transfer rate can be deduced from the transfer size, as discussed in the next section.

processes being swapped in.

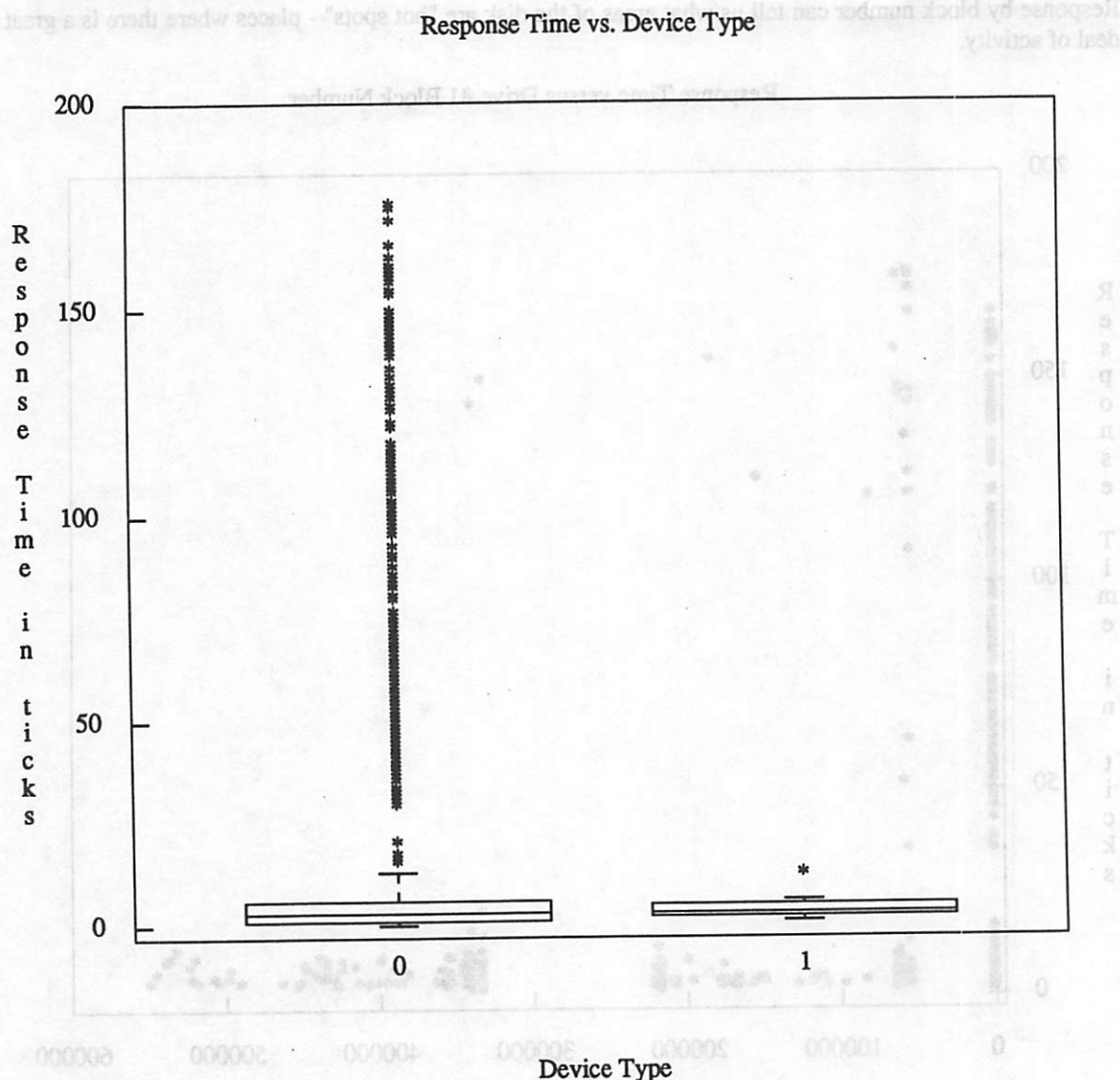


Figure 4. Response Time versus Device type (block=0,row=1)

We see from the data that transfers to and from the raw device experience good response times, and thus swaps occur relatively rapidly. Due to the rather small size of this data sample, we did not experience a swap request intermingled with data transfer caused by a "sync", where the queued buffers which have been written into are actually transferred to the disk<sup>14</sup>. Such a request may have shown extremely poor response time.

### 3.5 Transfer Size

The transfer size (in blocks) plotted against the response time gives some measure of the contribution of the device bandwidth versus the queueing delay experienced by a request. If the device speed was responsible for much of the delay, large requests should take much longer than smaller requests.

14. Update() is the kernel routine called when the system call sync() is executed by some process. /bin/sync is typically this process.

Response Time vs. Size in Blocks

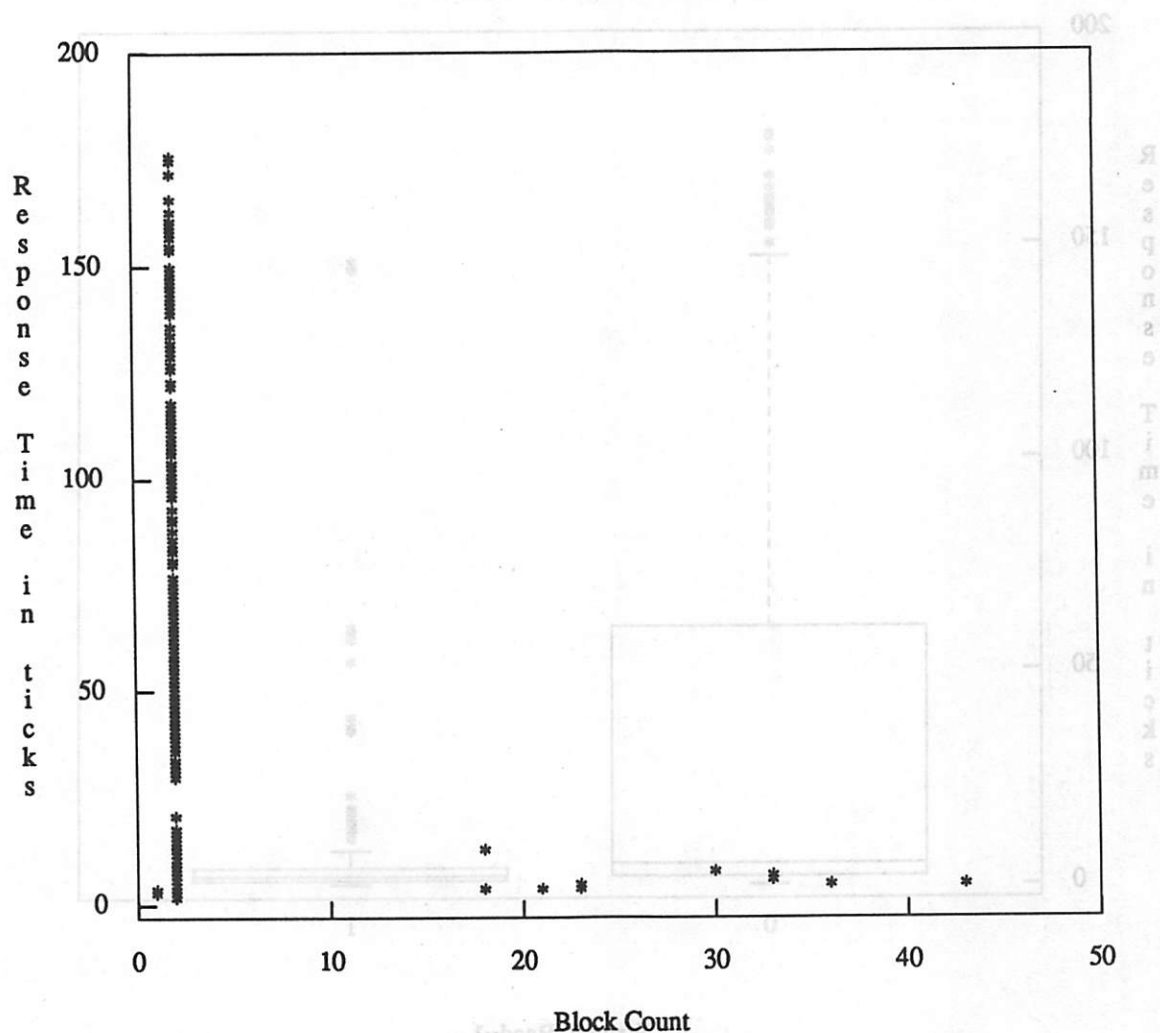


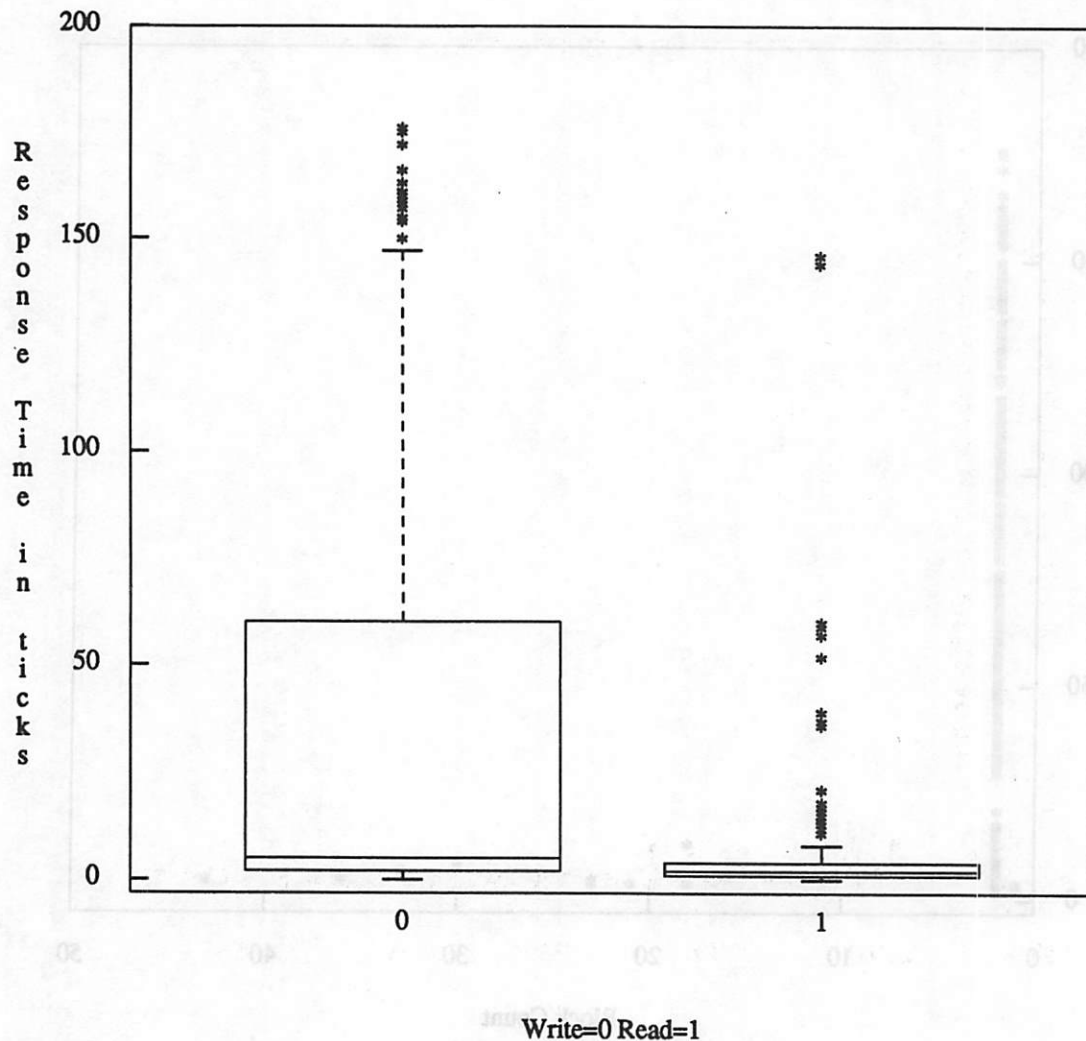
Figure 5. Response Time versus Transfer Size (blocks)

As can be seen from the graph, there seems to be little, if any, relation between transfer size and response time. This indicates that the time spent in the drive queue is more significant in its effect on response times than the size of the transfer (which is directly related to disk bandwidth).

### 3.6 Transfer Type

There are two transfer types, read and write. It is interesting to see what response times are experienced by these different types of requests.

### Response Time vs. Read/Write



**Figure 6. Response Time versus Transfer Type**

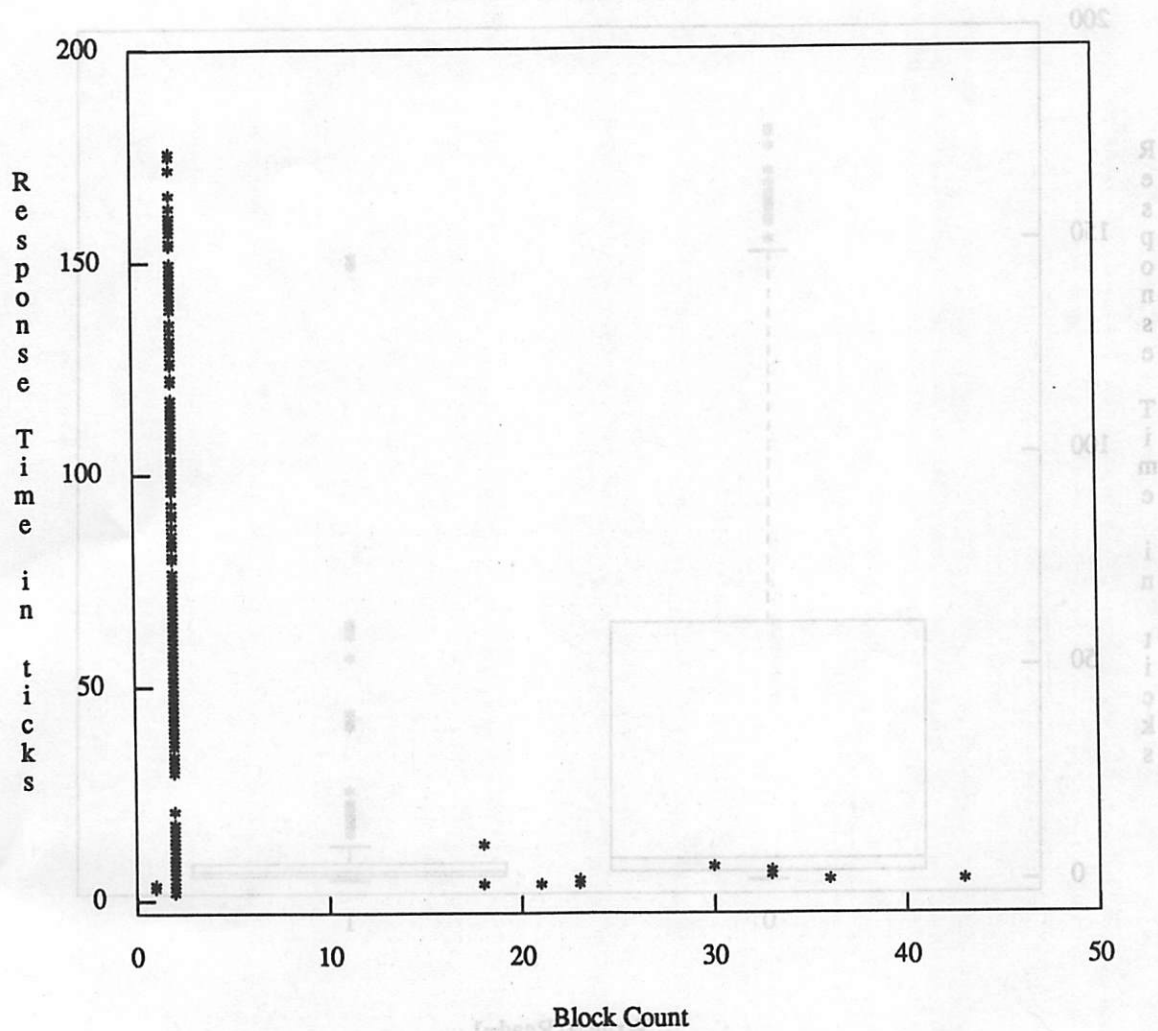
It seems clear from the graph that writes in general seem to experience a longer response time than reads, although there are a few reads which take a long time to complete. The explanation for this is quite simple: writes to block devices only happen as the result of buffers being flushed due to a shortage or due to a "sync" being executed. These both result in a large number of write requests being seen by the disk driver in a short time period. The congestion caused by this behavior results in many write requests exhibiting an extremely large response time. Some further analysis has shown that the read requests which exhibit large response times are those which are queued for the drive in a particular interval. The interval occurs shortly after a "sync" has been initiated. Thus, reads queued in this interval must wait for the large preceding queue of writes to be flushed<sup>15</sup>.

#### 3.7 Start Time

The response time of a request plotted against the start time can tell you if there are time-related events occurring which can affect the response time seen by disk requests.

<sup>15</sup>. Requests are not reordered by the driver software, as the hardware provides seek optimization.

### Response Time vs. Size in Blocks



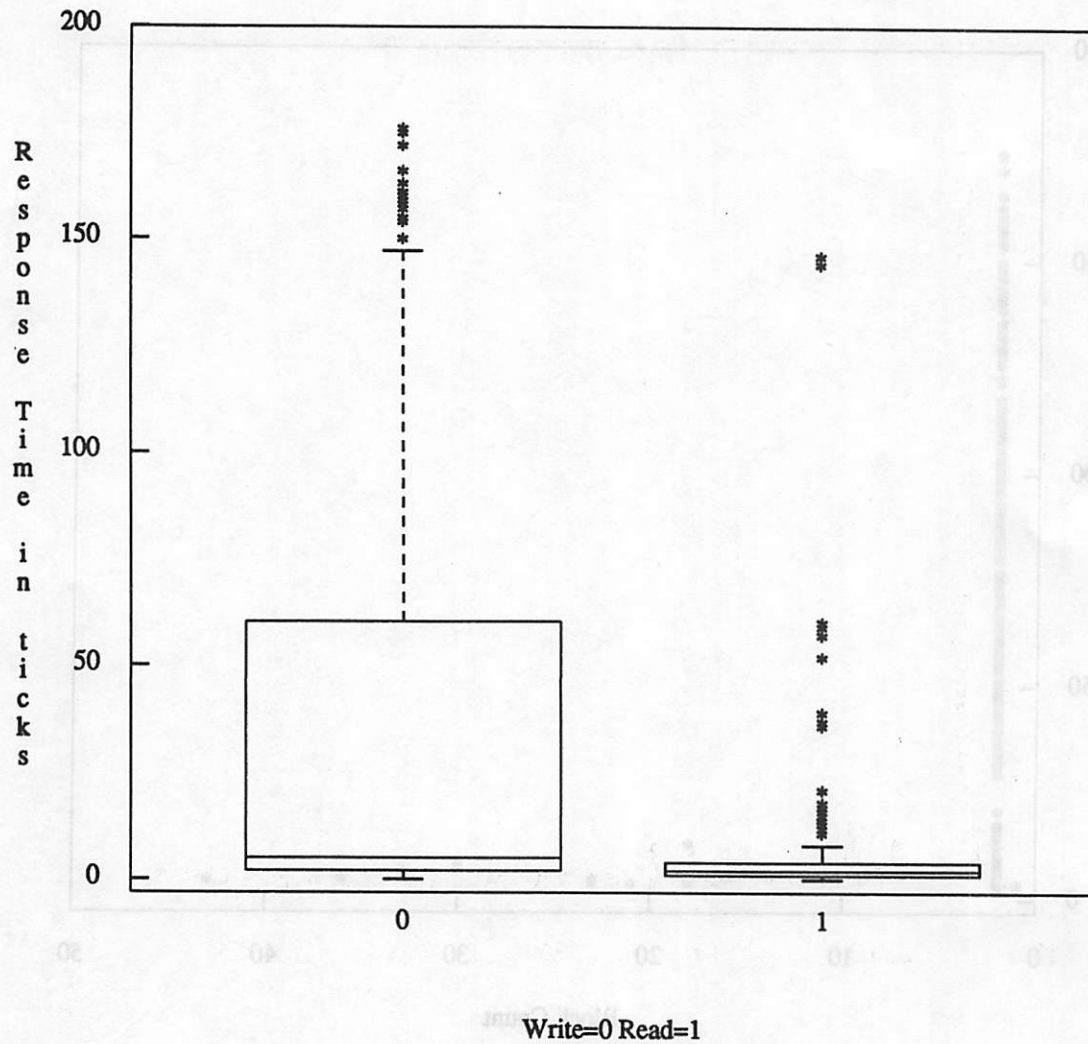
**Figure 5. Response Time versus Transfer Size (blocks)**

As can be seen from the graph, there seems to be little, if any, relation between transfer size and response time. This indicates that the time spent in the drive queue is more significant in its effect on response times than the size of the transfer (which is directly related to disk bandwidth).

#### 3.6 Transfer Type

There are two transfer types, read and write. It is interesting to see what response times are experienced by these different types of requests.

### Response Time vs. Read/Write



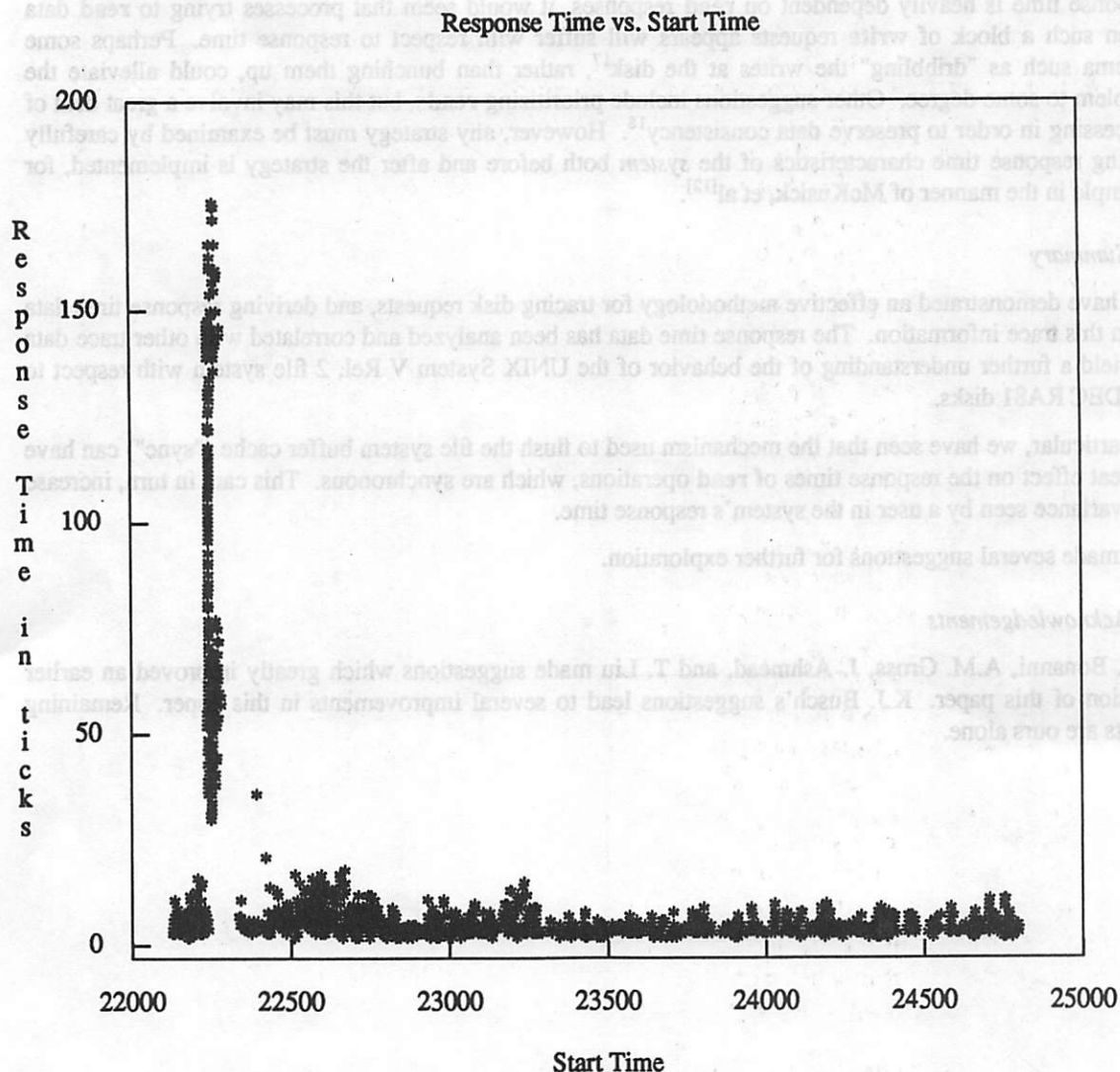
**Figure 6. Response Time versus Transfer Type**

It seems clear from the graph that writes in general seem to experience a longer response time than reads, although there are a few reads which take a long time to complete. The explanation for this is quite simple: writes to block devices only happen as the result of buffers being flushed due to a shortage or due to a "sync" being executed. These both result in a large number of write requests being seen by the disk driver in a short time period. The congestion caused by this behavior results in many write requests exhibiting an extremely large response time. Some further analysis has shown that the read requests which exhibit large response times are those which are queued for the drive in a particular interval. The interval occurs shortly after a "sync" has been initiated. Thus, reads queued in this interval must wait for the large preceding queue of writes to be flushed<sup>15</sup>.

#### 3.7 Start Time

The response time of a request plotted against the start time can tell you if there are time-related events occurring which can affect the response time seen by disk requests.

<sup>15</sup>. Requests are not reordered by the driver software, as the hardware provides seek optimization.



**Figure 7. Response Time versus Start Time**

This particular plot shows a window of about 3000 ticks of time; the time values were obtained from the least significant 16 bits of the system lbolt timer.

It seems obvious that there is some event which is causing the large "spike" in response time seen in the graph. This event is a "sync". Any requests which are queued in the same time frame as the "sync" are affected by it; in particular read requests are postponed until the preceding write requests are satisfied.

### 3.8 Conclusions

The data which we have analyzed have shown that the devices themselves are not particularly slow<sup>16</sup>. What seems to most affect the response time of read requests is their timing with relation to the large blocks of write requests generated by a "sync". Since writes are already asynchronous, and interactive

16. The observation must be made, however, that a timer with a resolution on the order of 20 milliseconds is inadequate for logging transactions taking place at the speeds of these disk transfers. For example, a relatively large percentage of requests exhibited response times of 0 ticks; it is clear that we are losing information.

response time is heavily dependent on read responses, it would seem that processes trying to read data when such a block of write requests appears will suffer with respect to response time. Perhaps some schema such as "dribbling" the writes at the disk<sup>17</sup>, rather than bunching them up, could alleviate the problem to some degree. Other suggestions include prioritizing reads, but this may involve a great deal of processing in order to preserve data consistency<sup>18</sup>. However, any strategy must be examined by carefully testing response time characteristics of the system both before and after the strategy is implemented, for example in the manner of McKusick, et al<sup>[12]</sup>.

#### 4. Summary

We have demonstrated an effective methodology for tracing disk requests, and deriving response time data from this trace information. The response time data has been analyzed and correlated with other trace data to yield a further understanding of the behavior of the UNIX System V Rel. 2 file system with respect to the DEC RA81 disks.

In particular, we have seen that the mechanism used to flush the file system buffer cache ("sync") can have a great effect on the response times of read operations, which are synchronous. This can, in turn, increase the variance seen by a user in the system's response time.

We made several suggestions for further exploration.

#### 5. Acknowledgements

L.E. Bonanni, A.M. Gross, J. Ashmead, and T. Liu made suggestions which greatly improved an earlier version of this paper. K.J. Busch's suggestions lead to several improvements in this paper. Remaining faults are ours alone.

17. For example, since the buffer cache is maintained as an LRU list, we can easily select some number of the "least recently written" blocks for queuing to the drive. This could be done with some mechanism such as the callout table which provides a way to schedule periodic activities inside the UNIX kernel.

18. Inconsistency can easily be generated by reordering disk requests, in particular, by putting a read in front of a write for the same block

## REFERENCES

1. *A Fast File System for UNIX*  
Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry  
Computer Systems Research Group Technical Report #7,  
Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
July 27, 1983
2. *Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX*  
Bob Kridle and Marshall Kirk McKusick  
Computer Systems Research Group Technical Report #8,  
Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
July 27, 1983
3. *The Design of the UNIX Operating System*,  
Maurice J. Bach  
Prentice-Hall  
1986
4. *UDA50 User Guide*  
EK-UDA50-UG-003  
Digital Equipment Corporation
5. *RA81 Disk Drive User Guide*  
EK-ORA81-UG-001  
Digital Equipment Corporation
6. *VAX Hardware Handbook* (1982-1983)  
Digital Equipment Corporation,  
Order Code EB-21710-20
7. *The UNIX System Activity Package*  
Tsyh-Wen Pao  
Bell Telephone Laboratories 3644-800313.01MF  
March 13, 1980
8. *The UNIX System Activity Package - UNIX 4.0*  
Tsyh-Wen Pao  
Bell Telephone Laboratories 45173-810106.01MF  
January 6, 1981
9. *S - An Interactive Environment for Data Analysis and Graphics*  
Richard A. Becker  
John M. Chambers  
Wadsworth Statistics/Probability Series  
1984
10. *Introduction to Statistics*,  
Herbert Robbins and John Van Ryzin,  
SRA, 1975
11. *Graphical Methods for Data Analysis*,  
John M. Chambers, William S. Cleveland, Beat Kleiner, and Paul A. Tukey  
Duxbury Press  
1983

12. *Measuring and Improving the Performance of Berkeley UNIX*

Marshall Kirk McKusick, Samuel J. Leffler, Michael J. Karels, Luis Felipe Cabrera

Computer Systems Research Group

Computer Science Division

Department of Electrical Engineering and Computer Science

University of California, Berkeley

November 30, 1985

# Tuning UNIX Lex or It's NOT True What They Say About Lex

Van Jacobson

*Real Time Systems Group, Lawrence Berkeley Laboratory,  
University of California, Berkeley, CA 94720*

van@lbl-rtsg.ARPA

## ABSTRACT

Unix Lex performance has been getting dismal reviews lately. At the Atlanta USENIX, Peter Honeyman [1] noted that Pathalias sped up a factor of two when its Lex scanner was replaced by a hand-coded scanner. Several USENET authors have written of similar experiences and the phrase "it's true what they say about Lex" has become rooted in Unix folklore. This folklore is now spreading to the Formal Language community: In recent articles on scanning, W. M. Waite [2] and V. P. Heuring [3] claim that poor performance is inherent in table-driven scanners and alternative scanner generators should be developed.

The Real Time Systems Group at LBL has used table-driven finite automata to construct some of the world's fastest data acquisition and control systems. We had doubts about the "inherently slow" claim and were curious why Lex went so slow. On investigation, we found a number of implementation problems but nothing that looked fundamental. To demonstrate there were no inherent problems, we attempted to write a "tuned" version of Lex. About one week's effort was invested in recoding the Lex runtime recognizer loop and making small changes in the table output routine. The result was a factor of ten performance improvement. When tested against the hand-coded scanners described in [1] and [2], the new Lex was always faster. There was also a serendipitous result: a new table compression scheme, intended to speed up the recognizer inner loop, resulted in a factor of two table size reduction. The new Lex scanners were not only faster than their hand-coded counterparts, both the original source and final machine code were smaller.

This paper describes the tuning work. In the next section we introduce a fragment of C that will be used as an ongoing example. We give an ad-hoc scanner for this fragment and two forms of table driven scanner, one using full tables and the other using S. C. Johnson's elegant compression algorithm [4,5] (this style of compressed table is used by both Lex and Yacc).

The performance of these scanners is compared in section three. The time to execute each scanner was computed using Knuth's time honored method of counting MIX instructions. Since instruction counting gives only execution time per type of token, it was also necessary to know the token type and length distributions for typical inputs. We give these distributions and show how they were produced by running small Lex and Awk programs over C source files.

In section four we investigate why the original Lex is slow compared to the "ideal" table driven scanner of section one. Extensive table compression accounts for about a third of the slow down. While compression was necessary in the original PDP-11 Lex, changing memory prices and cpu architecture have made it more costly than beneficial today. The remainder of the slow down is due to the cumulative cost of several infrequently used Lex features (e.g., YYREJECT, trailing context). We describe the impact of each of these features on the recognizer loop.

Section five describes the tables and runtime of the new Lex and the changes made to the original Lex source. We also describe how adding a little intelligence to the Lex compiler saves the user from paying the cost of unused features: While compiling, the new Lex keeps track of what features are used by the current input file and produces a scanner loop tailored for only those features. Finally we discuss the breakdown of the tuning work: In particular, why it took only three hours to modify the 600 lines of code involved in Lex table output but five days to write the six line subroutine that replaced Lex's 200 line recognizer.

- [1] Peter Honeyman, 'PATHALIAS or The Care and Feeding of Relative Addresses', *The 1986 Summer USENIX Proceedings*, June, 1986, p.126.
- [2] W.M. Waite, 'The Cost of Lexical Analysis', *Software-Practice and Experience*, 16(5), May, 1986, p.473.
- [3] V.P. Heuring, 'The Automatic Generation of Fast Lexical Analysers', *Software-Practice and Experience*, 16(9), September, 1986, p.801.
- [4] A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977, Sect. 3.8.
- [5] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley, 1986, Sect.3.9.

## Methodology and Results of Performance Measurements

### for a New UNIX Scheduler

Jeffrey H. Straathof<sup>1</sup>, Ashok K. Thareja<sup>1</sup>, Ashok K. Agrawala

Department of Computer Science  
University of Maryland  
College Park, MD 20742

#### ABSTRACT

A paper presented at the Winter 1986 USENIX Conference (Straathof, et. al., 1986) described motivation, design, and implementation of a new process scheduler developed at the University of Maryland for UNIX<sup>2</sup> 4.3BSD. The new scheduler, ESCHED, has now been completely implemented, debugged and tested. We have also conducted experiments to compare the performance of ESCHED against the existing 4.3BSD process scheduler. This paper describes the methodology of these experiments as well as their results.

In order to conduct the experiments, we developed a Remote Terminal Emulator (RTE). The RTE permitted emulating activities of a number of terminals as if they were actually connected to a system. Using a PYRAMID 90x as the RTE and a VAX 11/750 as the System Under Test (SUT), the scheduler was tested under different workloads. The RTE was implemented so that terminal emulation could be performed over direct serial lines and over an Ethernet running TCP/IP.

Using the RTE, a series of experiments was conducted to measure the performance of ESCHED. Four experiments, that demonstrate how ESCHED satisfies the four design goals, are reported in this paper. The first experiment measures and compares the interactive responsiveness of both the original and the new schedulers as system load is varied. The second experiment measures the impact on system throughput obtained by ESCHED. The third determines the effectiveness of the external mechanism provided by ESCHED for control of resource usage. The final experiment demonstrates the opportunities available for tuning of ESCHED.

<sup>1</sup> Present address: A&T Systems, Inc. 12904 Olivine Way, Silver Spring, MD 20904.

<sup>2</sup> UNIX is a trademark of AT&T Bell Laboratories.

## 1. Introduction

A paper presented at the 1986 Winter USENIX conference (Straathof, et. al., 1986) described motivation, design, and implementation of a new process scheduler developed for UNIX 4.3BSD at the University of Maryland. The new scheduler, ESCHED (an acronym for Event-driven SCHEDuler), was developed with these four explicit goals in mind: 1) interactive responsiveness, 2) throughput, 3) external control of resource usage, and 4) adaptability.

The ESCHED process scheduler (Straathof, 1986) employs a multilevel feedback run queue. Under its operation, a process in a higher level receives service before a process in any lower level, and an arriving process at any higher level will preempt a process running out of levels lower than that of the arriving process. The priority of a process directly determines the run queue level in which it will reside. A scheduled process retains control of the cpu until it voluntarily blocks, until it is preempted, or until its quantum of cpu service expires. The quantum size given to a scheduled process is determined by the run queue level from which it came; quantum sizes are usually small at the higher levels and large at the lower levels.

The priority of a process during operation of ESCHED is determined by the sequence of events the process completes. Completion of events that commonly identify interactive processes, e.g. terminal input, disk input, and socket input, increase the priority of a process. Completion of events that commonly identify cpu bound processes, e.g. quantum expirations, decrease the priority of a process. ESCHED attempts to keep interactive processes in the upper levels of the run queue and cpu bound processes in the lower levels, thus providing better service for the interactive processes.

External control during operation of ESCHED is provided with the ability to alter the maximum and minimum run queue levels within which a process is allowed to reside. Raising the range of a process increases its priority relative to other processes, and conversely, lowering the range decreases its priority. The initial maximum and minimum values of the *init* process, from which all user processes descend, provide default maximum and minimum priority values for all user processes.

ESCHED has been completely implemented, debugged and tested, and has been operational at the University of Maryland as with several other sites. We have also conducted experiments to compare the performance of ESCHED against the existing 4.3BSD process scheduler. In this paper, we present the methodology of these experiments and their results.

The next section describes in detail the design and implementation of an RTE developed for experimentation. Section 3 presents various experiments and their results. Section 4 includes some thoughts of retrospection and future directions.

## 2. A Performance Evaluation Tool

The need to evaluate the performance impact of ESCHED led to the development of a new evaluation tool. The development process began with an outlining of the requirements of the tool, including the design specifications. It concluded with the actual implementation. The following sections describe these steps.

## 2.1. Design Specifications

The approach taken was to develop a tool that could simulate every interaction between a typical set of users and a system. This approach was accomplished through the development of a RTE (Remote Terminal Emulator) ([Morris, 1982], [Calzarossa, 1984]).

Two machines are required to perform the benchmarking of a system when the RTE approach is taken. One machine runs several drivers that read scripts of user commands, feeds the commands to the system under test (SUT) as users would, and then records the response time value for each of the commands. The second machine, the one being benchmarked, reads the commands from the first machine as though the commands were coming from users' keyboards. It then executes the commands and sends the responses back to the first machine as though they were going to the users' displays. The first machine can obtain a response time value for each of the commands as a difference in time between the sending of the command and the receiving of the response.

Communication between machines is possible by connecting tty ports from one machine to tty ports of the other. The system under test operates just as though terminals were connected to its tty ports. Communication is also possible via remote logins over a network to which both machines are connected. A script driver writes commands and reads results through either a connected tty port or via remote login processes.

Trace files can be created on the first machine, the RTE, as an aid to show that the drivers actually performed the proper test. Results files can also be created containing a response time for each of the commands executed by the second machine, the SUT. The response time values can then be used as input into data reduction programs that can provide meaningful summary information indicating the performance of the SUT. Counts of the response time values and the amount of time required to complete the test can provide throughput measurements, since each response time represents a request, and throughput is simply a measure of requests completed per unit time. Figure 1 displays the components required to perform remote terminal emulation.

## 2.2. Implementation

The implementation of the RTE included script drivers, a controller of the script drivers, the script files, the trace files, and the result files. Programs were also created to measure the cpu utilization on the SUT, and to support remote logins so that script drivers could communicate to the SUT via a network. The implementation made use of several existing programs that helped provide remote login service. The following sections describe some of the parts involved in the implementation. It is assumed that the reader is familiar with the UNIX programming environment.

### 2.2.1. The Script Driver

The heart of the RTE resides in the script driver. It is the script driver that sends a command to the SUT, waits for the response, records the time interval, and makes a trace of all transactions. The script driver was implemented to be both efficient and functional.

### 2.2.2. The Script File

The script file contains both commands to control execution of the driver and commands that should be sent to the SUT for execution. An example of a script file read by a driver is shown in Figure 2. A description of the example follows.

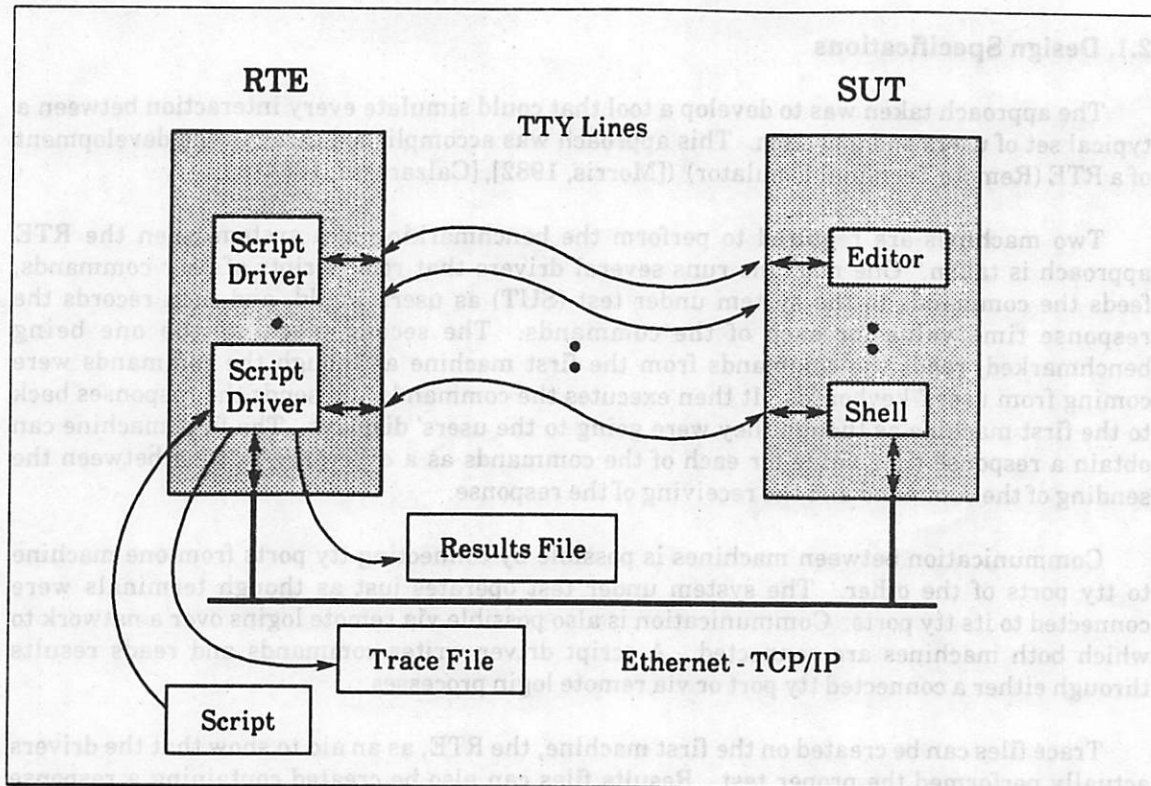


Figure 1. RTE Components

The first items in the script file inform the driver of the number of command classes possible in the script file and the range of think time values for each class. Think times occur between each pair of consecutive commands sent to the SUT and are uniformly distributed between the range.

The *.group* and corresponding *.endgroup* commands instruct the driver to execute the nested commands, in this case, one time. The pair of commands can be nested within another pair.

The *.class* command informs the driver that it is collecting response times for class 0. The class identifier is attached to each response time value written to the results file. This permits response time results to be produced for various classes of commands.

The script driver will not send a command to the SUT until the response from the previous command is complete. The *.wait* command instructs the driver to assume that a response is complete after three seconds of silence on the communication link from the SUT. This is the first of two ways the script driver will know that a response has completed. The second way, with use of the *.look* command, instructs the driver to search for a pattern of characters signifying the end of the response. In the example, the *.look* command instructs the driver to assume a response is over when it sees the next shell prompt.

The *.echo* command instructs the driver to assume that the SUT will echo the commands sent. The driver will then not consider the echoing as part of the response. The response will be determined when the first character after the echoed string is sent. The *.noecho* command

instructs the driver that the SUT will not echo the commands sent, useful primarily when editing files from within the script.

The string *jeff*\n is a command to be sent to the SUT, and in this case is the login id of a user. The lines not beginning with a recognized dot keyword are assumed to be strings sent as commands to the SUT. All of the commands sent from this script file end with a newline character.

*Realpasswd*, *set*, *date*, *users*, *ls*, and *logout* strings are all the beginnings of commands sent to the SUT. Their respective response time values will be recorded in the results file. Upon completion of the script, or with notification from the master script driver described later, the driver terminates.

### 2.2.3. The Trace File

The script driver can optionally produce a trace file during execution. The trace file contains the commands controlling execution of the driver, commands sent to the SUT for execution, responses received from the SUT, response times for the commands, and think times used between each sending of a command. An example of a portion of the trace file produced from the execution of the example script file in Figure 2 is shown in Figure 3. The content of the trace file is self explanatory.

```
1
5 20
.group 1
.class 0
.wait 3
.echo
jeff\n
.noecho
realpasswd\n
.echo
set prompt=[xxx]\n
.look [xxx]
date\n
users\n
ls\n
.wait 3
logout\n
.endgroup
```

Figure 2. Script File Example

```
> Sending...
set prompt="[xxx]\n"
> Echoing...
set prompt="[xxx]"
> Receiving...
[xxx]
> Response time 0.0800
> Think time 1
> .look [xxx]
> Sending...
date\n
> Echoing...
date
> Receiving...
Fri Jul 25 10:01:25 EDT 1986
[xxx]
> Response time 0.2100
> Think time 14
> Sending...
users\n
> Echoing...
users
> Receiving...
jeff neal pete rogerp rogerp
[xxx]
> Response time 0.2700
```

Figure 3. Trace File Example

#### 2.2.4. The Network Driver

The network driver is a program created to allow execution of a script driver that will conduct communication with the SUT via a remote login over a network as opposed to through a cable connecting tty ports. The network driver is identical to the driver just described, except that standard input and standard output are redirected to remote login processes that perform the communication with the SUT (see `rlogin(1)`).

#### 2.2.5. The Master Script Driver

The master script driver is a program created to control the execution of multiple script drivers run concurrently to perform one test. It reads a configuration file detailing information about the script drivers to be executed and executes them. Optionally, the master script driver can be informed to begin each script driver after a pause, and to stop all of the drivers after a specified period has elapsed.

The parts just described work together to form the RTE benchmarking tool. The next section describes the benchmarks performed to evaluate the effectiveness of ESCHED. The results of the tests are also presented.

### 3. Results of an Evaluation

A series of experiments was conducted to measure the performance of ESCHED. In particular, four experiments were conducted to determine whether or not ESCHED satisfies the four scheduling requirements outlined in section 1. The first experiment measured and compared the interactive responsiveness of both the original and new schedulers as system load varied. The second experiment measured the impact on system throughput obtained by ESCHED. The third determined the effectiveness of the external mechanism provided by ESCHED for control of resource usage. The final experiment demonstrated the opportunities available for tuning of ESCHED.

#### 3.1. Hardware Configuration

The same hardware configuration was used for all of the experiments. The machine that performed the benchmarking, the RTE, was a Pyramid 90x running OSx<sup>3</sup>. The machine benchmarked, the SUT, was a DEC VAX 11/750 running UNIX 4.3BSD with 8MB real memory and 2 Emulex drives. Six tty ports on the RTE were connected to six tty ports on the SUT providing one means of communication between the two machines. Both machines were connected to an Ethernet running TCP/IP that supported remote logins, i.e. the second means of communication.

The machines were part of a network supporting several dozen other machines. The benchmarking was conducted during periods when network traffic was very light so that contention for use of the network would not significantly affect the response time measurements. Additionally, commands sent to the SUT over the network were ones that

<sup>3</sup> OSx is a trademark of Pyramid Technology Corporation

<sup>4</sup> `Cron(8)` executes commands that need to be executed at specified dates and times.

had the largest expected response times. This was done to make the portion of a response time resulting from network contention relatively small.

All but one of the system daemons remained active during the experimentation. *Cron(8)*<sup>4</sup> was deactivated because of its tendency to irregularly increase system load significantly for short periods. The system load attributable to *cron(8)* would have been significantly different for each of the benchmarks. The additional load would then adversely affect the ability of the SUT to complete the commands being measured by the RTE, thus the obtained measurements would not be reflective of the assumed workload.

### 3.2. Script Selection

In order to measure the relative performance of various types of commands executed by the SUT, the commands were divided into three classes. Class 0 commands were very interactive ones and consisted of only simple full screen editing commands within *emacs(1)* and *vi(1)*. Example commands included: changing the cursor location, inserting text, deleting text, and searching for text.

Class 1 commands were somewhat interactive and were usually typed into a shell. Their response times were expected to be greater than those of class 0 commands, since significantly more resources were required to complete them. Example commands included: *grep(1)*, *date(1)*, *mail(1)*, *stty(1)*, *who(1)*, *cp(1)* and *wc(1)*. Command selection was made based on information obtained from process accounting data (see *acct(2)* and *lastcomm(1)*) and output of the *ps(1)* command.

Class 2 commands were not interactive, were considered cpu bound, and were always typed into a shell. Their response times were expected to be much greater than those of the class 0 and 1 commands, since they required even more resources to complete. These commands are run by typical users in the background since they do not normally complete quickly. Commands included only *nroff(1)*'s of several-page manual entries and *cc(1)*'s of files each containing several hundred lines of source code.

The command classes 0, 1 and 2 were used to distinguish users as very interactive ones, somewhat interactive ones, and non-interactive ones, respectively. Once logged into the SUT, each simulated user active during a benchmark would execute commands only of its associated class. Each user itself was classified into one of the same three classes, 0, 1 and 2, corresponding to the class of commands it executed after the login sequence. For example, one script driver simulating a class 0 user during a benchmark sent a login id, a password, a *vi(1)* command, and then a series a class 0 editing commands repeated until the benchmark ended.

Except for a few instances, all script drivers executing during a benchmark determined the end of each response with use of the *look* command, as described in section 2.3.2.. Script drivers simulating class 1 and class 2 type users looked for the next shell prompt to indicate the end of the previous response. Script drivers simulating class 0 type users looked for predetermined escape character sequences that are normally sent to terminals to control cursor movement within an editor.

Think times simulated between commands by the script drivers were uniformly distributed as described in section 2.3.2. The think time ranges for the three classes remained constant during all of the experiments. The range values are displayed in Table 1.

Script drivers were started by the master script driver ten seconds apart at the beginning of each test. This was done to prevent the SUT from experiencing an extremely high short-term load caused by having multiple users log into the system at the same time. Such a high

load would abnormally increase the response time values collected by the drivers until the system load could stabilize.

### 3.3. Interactive Response Time Evaluation

The primary goal of ESCHED was to maintain a stable and small interactive response time as system load varied. The goal of the first experiment was to determine the ability of ESCHED to maintain such an interactive response time.

#### 3.3.1. Experiment Description

For this experiment, the benchmarking tool simulated a fixed number of class 0 users while the number of class 1 and class 2 users were increased. The tests were initially performed while the SUT executed the original process scheduler. Identical tests were then performed while the system was running ESCHED. All of the script drivers simulating class 0 users communicated with the SUT via the linked tty ports. All of the script drivers simulating class 1 and class 2 users communicated via the network. Table 2 shows the combinations of users simulated during this phase of the evaluation.

Class	Low Value (seconds)	High Value (seconds)
0	0	2
1	10	20
2	30	60

Table 1. Think Time Range Values

Scenario	Class 0	Class 1	Class 2
a	6	0	0
b	6	1	1
c	6	2	2
d	6	4	4
e	6	6	6
f	6	8	8
g	6	10	10

Table 2. Number of Users per Scenario

Each scenario was executed for one hour with the original scheduler operating and one hour with the new scheduler operating. In order to ensure that the SUT was experiencing various loads, and to ensure that the SUT was well saturated with cpu bound activity during some of the scenarios, measurements were taken to indicate the cpu utilization of the SUT. Table 3 presents the percentages for the seven scenarios. During the benchmarking, periodic values of the BSD-style ten-minute load average were obtained to generate approximate values for the scenarios. These values are shown in Table 4 and are to be used only as an aid to understanding the load conditions placed on the SUT. The values were similar for benchmarking conducted with each of the two schedulers so only one average per scenario is presented.

The tunable parameters of ESCHED were fixed for most of the experimentation. Their values were chosen based solely on the experience obtained during the development of ESCHED. The chosen set of parameters made the scheduler on the SUT previously behave in a way that satisfied most of the users of that system. The parameter values are presented in Tables 5 and 6.

Scenario	Original Scheduler	New Scheduler
a	35.8	33.6
b	68.6	67.7
c	89.0	86.1
d	98.8	99.0
e	96.8	97.1
f	98.0	98.0
g	97.9	98.0

Table 3. CPU Utilization

Scenario	Load
a	0.5
b	1
c	2
d	4
e	8
f	12
g	17

Table 4. Approximate Values of the Load

### 3.3.2. Results

Response time values for the commands executed while the original scheduler was operating were collected for each scenario. Response time values were then collected while ESCHED was operating. Average values were calculated per class and are presented in Tables 7 and 8.

An examination of the values shows that ESCHED maintained a much smaller interactive response time, i.e. for class 0 commands, as the load on the SUT increased. The interactive response time values with operation of the original scheduler increased from 0.09 seconds to 0.69 seconds. The corresponding values with operation of ESCHED increased from 0.10 seconds to only 0.21 seconds. In the scenario with the greatest load, the value

Parameter	Value
Priority range of init process	1, 30
Priority range of swapping process	31, 31
Priority range of page daemon	31, 31
Terminal input priority increase (any mode)	4
Process birth priority increase	0
Block input priority increase	1
Socket input priority increase	1
Receipt of kill signal priority increase	31
Process death priority increase	31
Quantum expiration priority decrease	2

Table 5. ESCHED Parameters

Run Queue Levels(s)	Size
31	10
30-26	3
25-21	4
20-17	6
16-13	8
12-9	10
8-3	15
2-0	20

Table 6. ESCHED Quantum Sizes (hundredths of a second)

Scenario	Class 0	Class 1	Class 2
a	0.09	n/a	n/a
b	0.10	1.56	94.77
c	0.12	2.02	125.97
d	0.15	3.41	382.36
e	0.27	4.01	698.29
f	0.41	4.78	966.38
g	0.69	6.43	1277.76

Table 7. Average Response Times with Operation of the Original Scheduler

Scenario	Class 0	Class 1	Class 2
a	0.10	n/a	n/a
b	0.11	1.39	91.82
c	0.12	1.44	122.81
d	0.14	1.83	358.70
e	0.16	2.21	517.75
f	0.18	2.56	1137.30
g	0.21	3.72	1843.72

Table 8. Average Response Times with Operation of ESCHED

maintained by ESCHED was a third of the corresponding value maintained by the original scheduler.

While maintaining a smaller average response time for class 0 commands, ESCHED also maintained a smaller average response time for class 1 commands, i.e. those that were somewhat interactive. For several of the scenarios, the values maintained by ESCHED were close to half those maintained by the original.

ESCHED maintained response time averages for class 2 commands, i.e. those that are not interactive, in scenarios *b* through *e* that are comparable to those maintained by the original scheduler. Once the load increased beyond saturation of the cpu, scenarios *f* and *g*, the class 2 response time became markedly greater. In the scenario with the greatest load, the value maintained by ESCHED was almost 50 percent greater than the corresponding value of the original scheduler.

As can be observed by the values of scenario *g*, ESCHED is shifting the available processing power from the cpu bound process to the interactive processes. ESCHED, as was intended, maintained a much smaller interactive response time as system load increased.

### 3.4. Throughput Evaluation

An important goal of ESCHED was to increase system throughput by decreasing scheduling overhead. The goal of the second experiment was to determine the change in scheduling overhead obtained by ESCHED.

#### 3.4.1. Results

Tables 9 and 10 present the number of commands completed during the execution of the seven scenarios described in the previous sections. These tables, in combination with the average response time tables, show several results.

Scenario	Class 0	Class 1	Class 2
a	18956	n/a	n/a
b	18380	204	19
c	18190	360	33
d	17561	630	29
e	16452	937	25
f	13744	1082	23
g	10929	1267	11

Table 9. Number of Completed Transactions with Operation of the Original Scheduler

Scenario	Class 0	Class 1	Class 2
a	18351	n/a	n/a
b	18256	204	19
c	18079	368	33
d	17779	694	32
e	18260	1049	39
f	17887	1023	21
g	16926	1451	12

Table 10. Number of Completed Transactions with Operation of ESCHED

On a lightly loaded system of just interactive processes, scenario a, and under the chosen parameters for ESCHED, the new scheduler created slightly more overhead than operation of the original scheduler. The system operating the original scheduler was able to complete more commands, with a corresponding smaller average response time, than the system operating ESCHED. This ability to complete more commands in the hour of benchmarking

denotes a system that has better throughput, since throughput is measured in completions per unit time.

As the load increased by the addition of one class 1 user and one class 2 user in scenario *b*, the throughput difference between the two schedulers is almost negligible. The values in Tables 9 and 10 for scenario *b* show that the two schedulers completed the same number of both class 1 and class 2 commands. The original scheduler was able to complete approximately one percent more class 0 commands in the one hour benchmark. Because more commands were completed, and because the scheduler was the only change made to the system between tests, the original scheduler operated with less overhead and therefore with a slightly higher throughput.

During the execution of scenarios *c* through *e*, ESCHED clearly operated with better throughput. In all of the scenarios, and for all three command types, ESCHED was able to execute more commands and usually with smaller average response times. In scenario *e*, ESCHED completed 11% more class 0 commands, 12% more class 1 commands, and 56% more class 2 commands. Through these load ranges and with the chosen tunable parameters, ESCHED completed a significantly greater number of commands and therefore had better throughput.

From the data available for scenario *f*, it cannot be determined which scheduler is operating with better throughput. Operation of the original scheduler yielded more completions in the class 1 and class 2 categories, but not in the class 0 category. Since there is no direct relationship between the amount of work required to complete commands from any two different classes, the total amount of work completed for a scenario cannot be computed. Since neither scheduler operates with better throughput in all three classes, and there is not a way to compute a total value for work completed, throughput comparisons cannot be made.

The data presented for scenario *g* in Tables 7 through 10 have an interesting characteristic. Operation of ESCHED yields a greater number of commands executed in all three classes, yet it also yields an average response time for class 2 commands 44% greater than that under operation of the original scheduler. The new scheduler completed more work in scenario *g* and therefore had a better throughput, but it completed more class 2 work at the expense of a greater average response time. Further analysis should include a closer examination of the distribution of response time values for the class 2 commands, perhaps revealing more information pertinent to the understanding of the scenario. It should also be noted that the RTE does not count the partial completion of a command by the SUT, and the original scheduler might have produced more of these partial completions than ESCHED had.

The original scheduler had higher throughput than ESCHED under light load conditions, albeit the difference was small. The new scheduler showed a much larger increase in throughput as the load of the system increased, as was exemplified in the comparisons of scenario *e*. More analysis, perhaps with different values for the tunable parameters of ESCHED, is required to obtain more complete throughput measurements.

### 3.5. External Control Evaluation

Another goal of ESCHED was to provide convenient and predictable mechanisms for the external control of resource usage. The goal of this experiment was to determine the effectiveness of the mechanisms provided by ESCHED.

A new user program, *remin(1)*, was provided with ESCHED that allowed execution of a process with a limited priority range. The experiment conducted involved the execution of class 2 commands that were run with priority ranges that only allowed the resulting

processes to reside in the lowest level of the cpu run queue. The processes were always preempted from control of the cpu by processes in any of the higher levels. Because the priority-min of *init* was 1, all normal priority processes operated at or above level 1, i.e. they did not operate in run queue level 0. The low priority class 2 commands should not have affected the performance of normal priority processes.

This experiment examined the effects on the average response times of class 0 and 1 commands when class 2 commands were executed at low priorities.

### 3.5.1. Experiment Description

For this experiment, the benchmarking tool simulated a fixed number of class 0, 1 and 2 users. Table 11 shows the combination of users simulated during this phase of the benchmarking.

Scenario	Class 0	Class 1	Class 2
h	4	8	8

Table 11. Number of Users

The tests were conducted while the SUT was executing only ESCHED. First, scenario *h* was executed with all of the class 2 commands running at normal priorities. Next, the same scenario was executed with the class 2 commands running at low priorities.

### 3.5.2. Results

Table 12 presents the average response time values for the various classes of commands completed during the two executions of the scenario just described. As can be observed from

Priority of Class 2 Users	Class 0	Class 1	Class 2
normal	0.16	2.53	829.76
low	0.15	2.28	1019.94

Table 12. Effects of Adjusting Priorities on Average Response Times

the table, the average response times for the class 0 and class 1 commands decreased as the class 2 processes were executed at low priority. The control mechanism provided by ESCHED did effectively reduce the resource consumption of the class 2 commands, thereby making more of the cpu resource available for class 0 and 1 commands that could in turn produce responses more quickly.

### 3.6. Adaptability Evaluation

The final goal of ESCHED was to allow proper adaptation to the workload in which it might operate. The goal of this last experiment was to demonstrate that the changing, or

tuning, of the scheduling parameters can alter the performance of the scheduler in such a way that it will operate more desirably. The experiment was not conducted to show which parameters are optimum for a particular workload.

### 3.6.1. Experiment Description

This experiment involved execution of scenario *h* with some of the tunable parameters of Tables 5 and 6 set to new values. The new parameters were set with the intention of shifting some of the cpu processing power away from the class 0 and class 1 commands of scenario *h* to the class 2 commands. Table 13 shows the new values for the changed parameters.

Parameter	Old Value	New Value
Terminal input priority increase (any mode)	4	3
Quantum expiration priority decrease	2	5
Run queue levels 1 & 2 quantum sizes	20	25

Table 13. Tuned Scheduler Parameters

### 3.6.2. Results

The scenario was executed for one half hour before and after making the parameter changes. The results are presented in Tables 14 and 15. The average response time values for the class 0 and class 1 commands decreased as was expected. The average response time for the class 2 commands also decreased, as was not expected. An examination of Table 15 shows that the number of class 2 commands increased by one, so the class 2 commands did receive more processing power. An examination of the data revealed that the unexpected increase in the average response time was the result of a large response time for the one additional class 2 command executed.

Parameter Selection	Class 0	Class 1	Class 2
Untuned	0.16	2.53	829.76
Tuned	0.18	2.76	903.02

Table 14. Average Response Times Before and After "Tuning"

The benchmark did show that processing power could be shifted between classes of commands. A substantial amount of benchmarking would be required before an optimum set of tunable parameters for a specific workload executed on a specific system could be found, this because of the large number of possible permutations. Additional testing could reveal the relative effectiveness of changes in different parameters and possibly lead to the development of heuristics accurately predicting the performance of a system with a particular workload and tuned scheduler.

Parameter Selection	Class 0	Class 1	Class 2
Untuned	5991	705	9
Tuned	5798	670	10

Table 15. Number of Completed Commands Before and After "Tuning"

### 3.7. Summary of Results

As shown by results of the series of experiments conducted, ESCHED performed quite satisfactorily. Interactive response time remained acceptably small as system load increased; throughput was significantly higher for several of the scenarios executed; the control mechanism was effective; the parameters could be tuned to affect the distribution of the cpu resource. Though the benchmarking performed was by no means complete, it did clearly reveal that improved system performance is possible with a new UNIX 4.3BSD process scheduler.

### 4. Future Directions

The study of UNIX process scheduling is by no means complete. Examining any introductory operating system book will show that a variety of scheduling techniques currently exist that can be implemented, and there is always room for the development of new techniques.

As a starting point, additional benchmarking could be performed to better test the effectiveness of ESCHED. There exist a large set of permutations of the tunable parameters, and each permutation could be tested against a variety of workloads and on a variety of systems. Though testing the entire set of parameter/system/workload combinations would probably be very time consuming, additional testing might lead to heuristics that could reasonably predict the performance of a combination. The heuristics could then be used to tune the parameters so that a scheduler operates close to its optimum for a given system/workload combination. Once a much better understanding of the relationship between parameters and performance was obtained, a scheduler could be developed to dynamically tune itself.

Additionally, it would be interesting to see the effects of several modifications to ESCHED. This first possible modification involves the value of the quantum associated with a process as it begins receiving service from the cpu. As was described, a quantum is initialized to a value associated with the level from which the process left the run queue. If a process involuntarily releases control of the cpu, i.e. is preempted by a process with a higher priority that has become runnable, it resumes control at a later time with a full quantum. With some minor modifications to the implementation, the preempted process could resume control with only the remainder of the previous quantum. This change would make cpu bound processes move down the run queue more quickly, since some of the quanta would then be smaller and therefore expire sooner. This change would also reduce the throughput of the system, since more quantum expiration would occur producing more overhead, i.e. processing power that could be used to complete user requests.

It would also be interesting to see an additional scheduling layer placed above the scheduler described in this paper. The additional layer would schedule the cpu on a per user

basis; the user requiring cpu service who had received the smallest percentage of recent service would be scheduled next. Of his processes, the one with the highest priority according to methodology of ESCHED would next receive service. This two-layered approach to process scheduling would divide the cpu between  $n$  active users in such a way that each received one  $n^{\text{th}}$  of the current processing power. This would allow no user to greatly affect the service of other users, since he could at most obtain only one  $n^{\text{th}}$  of the service. This approach would also allow a user who only wanted to perform cpu bound activities to obtain his fair share of the available service.

Regarding the remote terminal emulator developed to evaluate the performance of the new scheduler, it would be interesting to determine its usefulness in measuring other system changes. Other system changes might include repartitioning of file systems, attachment of new disks, changing of the decision making portion of the swapping process, varying the amount of real memory, and varying the amount of swap space available. The remote terminal emulator could also be used to compare the performance of different machines, each capable of executing a common set of scripts.

## 5. References

- Bayly, R.G. *The Design of Multi-Access Benchmarks*, in **Benchmarking: Computer Evaluation and Measurement**, Nicholas Benwell, ed. Washington, D.C.: Hemisphere Publishing Corp., 1975.
- Calzarossa, M. et. al. *A Tool for Performance Comparisons of UNIX-Based Systems*, University of Pavia and Milano, Pavia, Italy, 1984.
- Morris, M. F. and P. F. Roth. **Computer Performance Evaluation: Tools and Techniques for Effective Analysis**, New York, NY: Van Nostrand Reinhold Co., 1982.
- Straathof, J. H. *ESCHED: A New UNIX Process Scheduler and its Performance*, Master's Thesis, University of Maryland. August 1986.
- Straathof, J. H., A. K. Thareja, A. K. Agrawala. *UNIX Scheduling for Large Systems*, in **Proceedings of the Denver USENIX Conference**, January 1986, pp. 111-139.

# News Need Not Be Slow

Geoff Collyer

Department of Statistics\*

University of Toronto

utzooolutcsri!utfraser!geoff

Henry Spencer

Zoology Computer Systems

University of Toronto

utzoool!henry

C news is a re-write, from scratch, of the 'transport layer' of the Usenet software. C news runs at over 19 times the speed of B news; C expire runs in minutes rather than the hours taken by B expire. These performance improvements were (mostly) quite simple and straightforward, and they exemplify general principles of performance tuning.

## 1. History and Motivation

In the beginning (of Usenet) (1979) was A news, written at Duke University by Steve Bellovin, Stephen Daniel, Tom Truscott and others. A single program, *news*, received, relayed, perused and cleaned out news articles. All articles were stored in a single UNIX<sup>†</sup> directory, which made A news suitable for local news and low volumes of network news. News articles were exchanged using a simple message format in which the first five lines of a message contained the information nowadays found in the article headers: unique article-id, newsgroup(s), author, subject, date posted.

As Usenet began to grow (1981), people in and around the University of California at Berkeley, including Matt Glickman and Mark Horton, modified A news extensively. The articles of each newsgroup were now stored in a separate directory. The message format was changed from the rigid and inextensible A news header format to one conforming to ARPA RFC 822 (the current ARPA mail-message format standard). *News* was broken into separate programs: *readnews*, *inews* (aka *rnews*), and *expire*. The authors dubbed the result "B news". Since the release of B news, it has replaced A news almost<sup>‡</sup> everywhere on Usenet.

\* Work done mostly while at U of T Computing Services.

† UNIX is a registered trademark of AT&T.

‡ AT&T Bell Laboratories Research still runs A news for local newsgroups.

It soon became clear that sending individual articles from machine to machine as separate *uucp* transactions was unacceptably slow, in part because it produced large *uucp* spool directories, which are searched quite slowly\* by the kernel. Sites began to *batch* articles into batches of (typically) 50,000–100,000 bytes for transmission to other machines.

At about this time, B news was changed to file news articles in a tree, as (for example) `/usr/spool/news/net/women/only/123`, rather than as `/usr/spool/news/net.women.only/123`. The motive for this was primarily elimination of problems with long newsgroup names, but shortening directories (and thus speeding searches) was also a consideration.

As Usenet traffic continued to grow explosively, sites began to use data compression on news batches. The main objective was to reduce expensive long-distance phone time, but again performance improved a bit: the extra processor time used for compression and decompression was more than gained back by the reduction in processor time used by *uucp* itself.

Unfortunately, B news has been modified by many people since 1981, and has mutated repeatedly to match the changing nature of Usenet. It has become complex, slow, and difficult to maintain.

During 1985, we observed that the nightly arrival of new news and expiry of old news were consuming resources far out of proportion to the volume of data involved†. *Expire* often ran for 90 minutes, and *rnews* processing averaged 10 seconds or more per article. Both programs tended to cripple systems by performing much disk i/o and eating much system-mode CPU time. *Uts* was running B 2.10.1 news then and *utzo* was running B 2.10 news. Although newer B news releases were available, they offered little beyond B 2.10, and it was often necessary to regression-test new B news releases to verify that reported, published bug fixes had in fact been applied.

Spencer acted first and rewrote *expire* from the ground up. Though it initially lacked any form of selective expiry, this *expire*, when run each night, finished in about 15 minutes. (This was on 750-class machines holding all Usenet news and expiring after 14 days.)

Collyer observed in November 1985 that B *rnews*, upon receiving a batch of news, immediately *execed* a trivial unbatcher which copied each article into a temporary file and then forked and *execed* B *rnews* again. Such a technique is clearly overkill for articles averaging about 3,000 bytes each. Preliminary experiments failed to produce a modified B *rnews* that could unravel a batch without forking. Consultation with Rick Adams, the current B-news maintainer, revealed that this same technique remained in the upcoming B news release (variously B 2.10.3 or B 2.11). Within one week‡, a from-scratch C *rnews* prototype was working well enough to run experimentally on a 'leaf' machine receiving a subset of news.

This prototype version lacked a good many necessary amenities, and over the next eight months it was enhanced to bring it up to full functionality. It was also tuned heavily to improve its performance, since it was faster than B *rnews* but still not fast enough to make us happy.

Once the *rnews* newsgroup name matching routines were working, Spencer revised *expire* to add selective expiry, specified in a control file. Recently, we have also revised our old batcher heavily, largely to add capability but with an eye on performance.

---

\* Recent *uucps* (notably Honey DanBer) provide spool sub-directories, and recent 4BSD (4.3BSD and later) kernels provide linear (as opposed to quadratic) directory searching, both of which help this problem.

† Never mind the cost/benefit ratio.

‡ 40 hours, Collyer didn't have to work hard.

## 2. Rnews Performance

The basic objective of C news was simpler code and higher performance. This may sound trite, but note that performance was an explicit objective. That was important. *Programs will seldom run faster unless you care about making them run faster.*

'Faster' implies comparison to a slower version. Knowing the value of improvements, and assessing this in relation to their cost, requires knowing the performance of the unimproved version. Collyer kept detailed records of his work on *rnews*, so he could see how much progress he was making. See the Appendix for the final result. *To know how to get somewhere, you must know where you are starting from.*

The first functional C *rnews* ran at about 3 times the speed of B *rnews*. We had assumed that merely eliminating the fork/exec on each article would give a factor of 10 improvement, so this was disappointing. *Avoiding obvious performance disasters helps... but it's not always enough.*

Profiling, first with *prof(1)* and later with 4.2BSD's *gprof(1)*, and rewriting of the bottlenecks thus discovered, eventually brought the speed up to over 19 times the speed of B *rnews*. This required a number of write-profile-study-rewrite cycles. There is undoubtedly still a lot of code which could be faster than it is, but since profiling shows that it doesn't have a significant impact on overall performance, who cares? *To locate performance problems, look through the eyes of thy profiler.*

Collyer first experimented with using *read* and *write* system calls instead of *fread* and *fwrite*, and got a substantial saving. Though the usage of system calls in this experiment was unportable, the saving eventually lead him to rewrite *fread* and *fwrite* from scratch to reduce the per-byte overheads. This helped noticeably, since pre-System-V *fread* and *fwrite* are really quite inefficient. *If thy library function offends thee, pluck it out and fix it.*

At the time, C *rnews* was doing fairly fine-grain locking, essentially locking each file independently on each use. News doesn't need the resulting potential concurrency, especially when *rnews* runs relatively quickly, and the locking was clearly a substantial fraction of the execution time. C *rnews* was changed to use B-news compatible locking, with a single lock for the news system as a whole. *Simplicity and speed often go together.*

When sending articles to a site using batching, *rnews* just appends the filename of each article to a *batch file* for that site. The batch file is later processed by the batcher. In principle, batching is an option, and different sites may get different sets of newsgroups. In practice, few articles are ever sent unbatched, and most articles go to all sites fed by a given system. This means that *rnews* is repeatedly appending lines to the same set of batch files. Noticing this, Collyer changed C *rnews* to keep these files open, rather than re-opening them for every article\*. *Once you've got it, hang onto it.*

These two simple changes—coarser locking and retaining open files—cut system time by about 20% and real time by still more.

On return from Christmas holidays, after considerable agonizing over performance issues, Collyer turned some small, heavily-used character-handling functions into macros. This reduced user-mode time quite a bit. *A function call is an expensive way to perform a small, quick task.*

---

\* The price for this tactic is that the code has to be prepared for the possibility that the number of sites being fed exceeds the supply of file descriptors. Fortunately, that is rare.

*Rnews* was always looking up files by full pathnames. Changing it to *chdir* to the right place and use relative names thereafter reduced system time substantially. *Absolute pathnames are convenient but not cheap.*

Studying the profiling data revealed that *rnews* was spending a lot of time re-re-reading the *sys* and *active* files. These files are needed for processing every article, and they are not large. Collyer modified *rnews* to simply read these files in once and keep them in core. This change alone cut system time and real time by roughly 30%. *Again, once you've got it, don't throw it away!*

There is a more subtle point here, as well. When these files were re-read every time, they were generally processed a line at a time. The revised strategy was to *stat* the file to determine its size, *malloc* enough space for the whole file, and bring it in with a single *read*. This is a vastly more efficient way to read a file! *Tasks which can be done in one operation should be.*

At this point (mid-January 1986), *C rnews* was faster than *B rnews* by one order of magnitude, and there was much rejoicing.

In principle, the 'Newsgroups:' header line, determining what directories the article will be filed in, can be arbitrarily far from the start of the article. In practice, it is almost always found within the first thousand bytes or so. By complicating *rnews* substantially, it became possible in most cases to *creat* the file in the right place (or the first of the right places) in */usr/spool/news* before writing any of the article to disk, eliminating the need for temporary files or even temporary links. The improvement in system time was noticeable, and the improvement in user time was even more noticeable. *Prepare for the worst case, but optimize for the typical case.*

There are certain circumstances, notably control-message articles, in which it is necessary to re-read the article after filing it. *Rnews* originally re-opened the article to permit this. Changing the invocation of *fopen* to use the *w+* mode made it possible to just seek back to the beginning instead, which is *much* faster. This, plus some similar elimination of other redundant calls to *open*, reduced system time by over 30%. *Get as much mileage as possible out of each call to the kernel.*

Both scanning the in-core *active* and *sys* files and re-writing the *active* file are simpler if the in-core copies are kept exactly as on disk, but this implied frequent scans to locate the ends of lines. It turned out to be worthwhile to pre-scan the *active* file for line boundaries, and remember them. *When storing files in an unstructured way, a little remembered information about their structure goes a long way in speeding up access.*

We already had a *STREQ* macro, just a simple invocation of *strcmp*, as a convenience. As a result of some other experience by Spencer, Collyer tried replacing some calls of *strcmp* by a *STREQN* macro, which compared the first character of the two strings in-line before incurring the overhead of calling *strcmp*. This sped things up noticeably, and later got propagated through more and more of the code. String-equality tests usually fail on the very first character. *Test the water before taking the plunge.*

While looking at string functions, Collyer noticed that *strcmps* to determine whether a line was a particular header line had the comparison length computed by applying *strlen* to the prototype header. With a little bit of work, the prototypes were isolated as individual character arrays initialized at compile time. This permitted substituting the compile-time *sizeof* operation for the run-time *strlen*. *Let the compiler do the work when possible.*

At this point, profiling was turned off temporarily for speed tests. Profiling does impose some overhead. The speed trials showed that *C rnews* was now running at over 15 times the speed of *B rnews*.

After months of adding frills, bunting and B 2.11 compatibility\*, Collyer again returned to performance tuning in August 1986. The 4.2BSD kernel on *utcs* now included the 4.3BSD *namei* caches, which improve filename-lookup performance considerably. Unfortunately, considerations of crash recovery dictated some loss in performance: it seemed desirable to put batch-file additions out by the line rather than by the block. *Performance is not everything.*

*Gprof* revealed that newsgroup name matching was an unexpected bottleneck, so that module was extensively tweaked by adding register declarations, turning functions into macros, applying *STREQN* and such more widely, and generally tuning the details of string operations. The code that handled *sys-file* lines got similar treatment next. The combination cut 40% off user-mode time. *Persistent tuning of key modules can yield large benefits.*

Newsgroup matching remained moderately costly, and an investigation of where it was being used revealed two separate tests for a particular special form of name. It proved awkward to combine the two, so the testing routine was changed to remember having done that particular test already. *If the same question is asked repeatedly, memorize the answer.*

By this time, the number of system calls needed to process a single article could be counted on one's fingers, and their individual contributions could be assessed. At one point it was desirable for a *creat* to fail if the file already existed, so this was being checked with a call to *access* first. John Gilmore pointed out that on systems with a 3-argument *open* (4.2BSD, System V), this test can be folded into the *open*. The elimination of the extra name→file (*namei*) mapping cut both system time and real time by another 15%. (Note that this system *does* have *namei* cacheing!) *File name lookups are expensive; minimize them.*

The development system (*utcs*, a 750) is now filing 2-3 articles per second on average; *utfraser* (a Sun 3/160 with an Eagle disk) is typically filing 6-7 articles per second. *C rnews* runs over 19 times as fast in real time as *B rnews*, over 25 times as fast in system-mode CPU time, roughly 3.6 times as fast in user-mode CPU time, and over 10 times as fast in combined CPU times.

With one exception (see *Future Directions*), it now appears that very little can be done to speed up *rnews* without changing the specifications. It seems to be executing nearly the bare minimum of system calls, and the user-level code has been hand-optimised fairly heavily.

### 3. Expire Performance

The rewrite of *expire* that started this whole effort was only partly motivated by performance problems. Performance was definitely bad enough to require attention, but the *B expire* of the time also had some serious bugs. Worse, the code was a terrible mess and was almost impossible to understand, never mind fix. Early efforts were directed mainly at producing a version that would *work*; rewriting *expire* from scratch simply looked like the easiest route. Decisions made along the way, largely for other reasons, nevertheless produced major speedups.

The first of these decisions was a reduction in the scope of the program. *B expire* had several options for doing quite unrelated tasks, such as rebuilding news's history file. The code for these functions was substantial and was somewhat interwoven with the rest. *C expire* adheres closely to a central tenet of the 'Unix Philosophy': *a program should do one task, and do it well*. This may appear unrelated to performance, but better-focussed programs are generally simpler and smaller, reducing their resource consumption and making performance tuning easier (and hence more likely). In addition, a multipurpose program almost always pays some performance penalty for its generality.

---

\* And supposed B 2.11 compatibility, as those who remember the short-lived cross-posting restrictions will recall.

The second significant decision had the biggest effect on performance, despite being made for totally unrelated reasons. For each news article, the B news history file contained the arrival date and an indication of what newsgroups it was in. This is *almost* all the information that *expire* needs to decide whether to expire an article or not. The missing\* data is whether the article contains an explicit expiry date, and if so, what it is. B *expire* had to discover this for itself, which required opening the article and parsing its headers. A site which retains news for two weeks will have upwards of 5,000 articles on file. A few dozen of them will have explicit expiry dates. *But B expire opened and scanned all 5,000+ articles every time it ran!* This was a performance disaster.

We actually did not want to parse headers in *expire* at all, because the B news header-parsing code was (and is) complex and was known to contain major bugs. The performance implications of this were obvious, although secondary at the time. Header parsing is itself a non-trivial task, and accessing 5,000+ files simply cannot be made cheap. *Information needed centrally should be kept centrally.*

The C news history file has the same format as that of B news, with one addition: a field recording the explicit expiry date, if any, of each article. If no expiry date is present in the article, the field contains '-' as a placemark†. In this way, the header parsing is done *once* per article, on arrival. In fact, the extra effort involved is essentially nil, since *rnews* does full header parsing at arrival time anyway. *Rnews* had to be changed to write out the expiry date, and code which knew the format of the history file had to be changed to know about the extra field. Perhaps a dozen lines of code outside *expire* were involved.

A crude first version of C *expire*, incorporating these decisions in the most minimal way, ran an order of magnitude faster than B *expire*. Precise timing comparisons were not practical at the time, since the original motive for C *expire* was that B *expire* had stopped working completely, crippled by bugs in its header parsing. Later versions of B *expire* did cure this problem, but we were no longer interested in putting up slow, buggy software just to make an accurate comparison.

Further work on C *expire* mostly concentrated on cleaning up the hasty first version, and on incorporating desired features such as selective expiry by newsgroup. Selective expiry caused a small loss in performance by requiring *expire* to check the newsgroup(s) of each article against an expiry-control list. Here, *expire* benefitted from the work done to speed up the newsgroup-matching primitives of *rnews*, since *expire* uses the same routines. *If you re-invent the square wheel, you will not benefit when somebody else rounds off the corners‡.*

One improvement that was made late in development was in the format of the dates stored in the history file. B *rnews* stored the arrival date in human-readable form, and *expire* converted this into numeric form for comparisons of dates. Date conversion is a complex operation, and the widely-distributed *getdate* function used by news is not fast. Inspection of the code established that *expire* was the only program that ever looked at the dates in the history file. There is some potential use of the information for debugging, but this is infrequent, and a small program that converts decimal numeric dates to human-readable ones addresses the issue. Both C *rnews* and C *expire* now store the dates in decimal numeric form. *Store repeatedly-used information in a form that avoids expensive conversions.*

---

\* Recent versions of B news have made some attempt to redress this lack, but haven't gone as far as C *expire*. The discussion here applies to the B *expire* that was current at the time C *expire* was written.

† It would be possible to simply compute a definitive expiry date for an article when it arrives, and record that. This would eliminate the decision-making overhead in *expire*, but would greatly slow the response to changes in expiry policy. Since one reason to change policy is time-critical problems like a shortage of disk space, this loss of flexibility was judged unacceptable. It is better to leave the expiry decision to *expire* and concentrate on making *expire* do it quickly.

‡ A corollary of this is: *know thy libraries, and use them.*

Actually, *C expire* bows to compatibility by accepting either form on input, but outputs only the decimal form as it regenerates the history file. Thus, in the worst case, *expire* does the conversion only once for each history line, rather than once per line per run. *"If they hand you a lemon, make lemonade"*.

If *expire* is archiving expired articles, it may need to create directories to hold them. This is an inherently expensive operation, but it is infrequently needed. However, checking to see whether it is in fact needed is also somewhat expensive... and the answer is almost always 'no'. The same is true of checking to see whether the original article really still exists: it almost always does. (This cannot be subsumed under generic 'archiving failed' error handling because a missing original is just an article that was cancelled, and does not call for a trouble report.) Accordingly, *C expire* just charges ahead and attempts to do the copying. Only if this fails does *expire* analyze the situation in detail. *Carrying a net in front of you in case you trip is usually wasted effort.*

Archiving expired articles often requires copying across filesystem boundaries, since it's not uncommon to give current news and archived news rather different treatment for space allocation and backups. Copying from one filesystem to another can involve major disk head movement if the two filesystems are on the same spindle. Since head movement is expensive, maximizing performance requires getting as much use as possible out of each movement\*. *Expire* is not a large program, and even on a small machine it can spare the space for a large copying buffer. So it does its archiving copy operations using an 8KB buffer. *Buying in bulk is often cheaper.* Since 8KB accommodates most news articles in one gulp, there is little point in enlarging it further. *The law of diminishing returns does apply to buying in bulk.*

Since *expire* is operating on the history file at (potentially) the same time that *rnews* is adding more articles to it, some form of locking is necessary. Given that *expire* has to look over the whole database of news, and typically has to expire a modest fraction of the articles, it is a relatively long-running process compared to *rnews*. Contention for the history-file lock can be minimized by noting that *rnews* never does anything other than append to the file. So *expire* can leave the file unlocked while scanning it; the contents will not change. When (and only when) *expire* reaches end-of-file, it locks the news system, checks for and handles any further entries arriving on the end of the history file meanwhile, and finishes up. *Locking data that won't change is wasteful.*

After careful application of these various improvements, *C expire* is fast enough that further speedup is not worth much effort. However, an analysis of where it spends its time does suggest one area that might merit attention in the future. *Expire* rebuilds the history file to reflect the removal of expired articles. The history file is large. *Expire* must also rebuild the *dbm* indexing data base, since it contains offsets into the history file. This data base is comparable in size to the history file itself, and is generated in a less orderly manner that requires more disk accesses.

Much of the time needed for these operations could be eliminated if *expire* could mark a history line as 'expired' without changing its size. This could be done by writing into the history file rather than by rebuilding the whole file, and the indexing database would not need alteration. This would also permit retaining information about an article after the article itself expires, which would simplify rejecting articles that arrive again (due to loops in the network, etc.) after the original has expired. The history file should still be cleaned out, and the indexing database rebuilt, occasionally. *C expire* contains some preliminary 'hooks' for this approach, but to date full implementation does not seem justified: *C expire* is already fast enough. *Know when you are finished.*

---

\* As witness the progressive increase in filesystem block size that produced major performance improvements in successive versions of 4BSD.

## 4. Batchter Performance

The C batchter is descended from a very old version written to add some minor functionality that was not present in the B batchter of the time. It is small and straightforward, and contains only a couple of noteworthy performance hacks.

The batchter works from a list of filed articles, to be composed into batches. The list is by absolute pathname. All of these files reside in the same area of the system's directory tree, and referring to them with absolute pathnames every time implies repeatedly traversing the same initial pathname prefix. To avoid this, the batchter initially *chdirs* to a likely-looking place such as */usr/spool/news*. Thereafter, before using an absolute pathname to open an article, it checks whether the beginning of the pathname is identical to the directory where it already resides. If so, it strips this prefix off the name before proceeding. *If you walk the same road repeatedly, consider moving to the other end.*

The batchter's input is usually in fairly random order, with little tendency for successive files to be in the same directory. If this were not the case, it would be worthwhile for the batchter to actually move around in the directory tree to be closer to the next file.

The batchter used to copy data using *putc(getc())* loops. This has been replaced by *fread/fwrite* which is significantly faster, especially if using the souped-up *fread/fwrite* mentioned earlier. *If you need to move a mountain, use a bulldozer, not a teaspoon.*

## 5. Future Directions

The one improvement we are still considering for *rnews* is a radical revision of the newsgroup-matching strategy. Newsgroup matching still consumes about 18% of user-mode processor time. The key observation is that the information that determines which newsgroups go to which sites seldom changes. It would probably be worth precompiling a bit array indexed by newsgroup and site, and recompiling it only when the *active* file or the *sys* file changes in a relevant way. This would cut the newsgroup-matching time to essentially zero.

*Rnews* would be faster (and simpler) if 'Newsgroups:' and 'Control:' were required to be the first two headers (if present) of each article. At present *rnews* tries to find them before starting to write the article out, so that it can put the article in the right place from the start, but it has to allow for the possibility that vast volumes of other headers may precede them.

Hashing *active*-file lookups in *rnews* would be fun, but profiling suggests that it's not worthwhile unless the number of newsgroups is in the thousands.

When PDP-11's are truly dead on Usenet, the use of large per-process memories *may* allow further speedups to *rnews* by reading the entire batch into memory at once and writing each article to disk in a few *writes* (it can't easily be reduced to a single *write* because headers must be modified before filing).

One optimization we have *not* considered is re-coding key parts in assembler. C news already runs on five different types of machine. Use of assembler would be a maintenance nightmare, and probably would not yield benefits comparable to those of the more high-level changes.

## 6. Acknowledgements

Ian Darwin ran the very earliest alpha versions of *rnews* and gave helpful feedback. Mike Ghesquiere, Dennis Ferguson and others have run later versions and prodded Collyer to fix or implement assorted things. John Gilmore and Laura Creighton read and criticized an early alpha version of *rnews*.

## 7. Appendix: rnews Times

Measurements have been taken on a VAX 750 running 4.2BSD under generally light load, using a batch of 297,054 bytes of net.unix-wizards containing 171 articles and ~104 cross-postings. All times are in seconds per article.

time	real	user	sys	comments
85 Dec 6 00:54	4.68	0.3	1.29	B news rejecting all. (b.1.rej)
85 Dec 6 00:54	3.184	0.69	0.67	first timing trial; <i>profiling on</i> (c.1)
85 Dec 6 00:54	0.66	0.175	0.199	rejecting all (c.2.rej)
85 Dec 6 03:25	0.58	0.175	0.175	still rejecting all (c.3.rej)
85 Dec 6 23:46	9.058	0.631	2.251	B news using private directories, rejected 53 of the 171 articles as "too old" (b.2)
85 Dec 7 00:24	2.0 (est)	-	-	on a 10 MHz 68000 with slow memory and slow disk (crude timings) (c.darwin.1)
85 Dec 7 00:40	7.576	0.684	2.403	B news without the "too old" reject code and having cleared out history (b.3)
85 Dec 7 04:43	1.99	0.49	0.53	accepting the articles, using read and write for bulk copies (c.4)
85 Dec 7 06:10	2.261 (!)	0.497	0.449	optimized by less locking & keeping batch files open (c.5)
85 Dec 7 07:32	1.383	0.491	0.414	same as the last one, but with a lower load average (around 1.5) (c.6)
85 Dec 16 03:43	1.380	0.447	0.374	for calibration after misc. cleanup (c.7, c.8)
86 Jan 13 00:23	1.232	0.349	0.301	turned hostchar() into a macro (c.9)
86 Jan 13 04:26	1.36	0.333	0.242	using in-core active file, under heavy load (c.10)
86 Jan 13 08:24	1.94	0.349	0.253	using in-core sys file too, under heavy load. Re-run this trial! (c.11)
86 Jan 13 08:42	0.892 (!)	0.332	0.245	re-run at better nice. Not striking, except for real time. Was run in a large directory; ignore. (c.12)
86 Jan 13 08:59	0.861 (!)	0.333	0.212 (!)	re-run at good nice & in a small directory. Have beaten B news by <i>one order of magnitude</i> on real & sys times! Beat it by more than twice on user time. (c.13)
86 Jan 21 19:15	1.208	0.349	0.245	creat 1st link under final name, only link to make cross-postings; with HDRMEMSIZ too small (c.14)
86 Jan 21 19:57	0.728	0.318	0.193	previous mod, with HDRMEMSIZ of 4096 (c.15)
86 Jan 22 01:20	0.719	0.315	0.166	fewer opens (just rewind the spool file), but Xref(s): not working (c.16)
86 Jan 22 01:53	0.637 (!)	0.314	0.154 (!)	fewer opens fixed to spell Xref: right; Xref: not working (c.17)
86 Jan 22 04:00	0.874	0.325	0.174	fewer opens with Xref: fixed (times may be high due to calendar) (c.18)
86 Jan 22 05:45	0.694	0.309	0.159	under lighter load, times are better (c.19)
86 Jan 24 04:29	0.715	0.317	0.129 (!)	turn creat & open into just creat, under slightly heavy load (c.20)
86 Jan 24 06:06	0.628 (!)	0.288 (!)	0.129 (!)	reduce number of calls on index (by noting line starts at the start) and strncmp (via macro) in active.mem.c, but still profiling and writing stdout and stderr to the tty (c.21)
86 Jan 24 07:22	0.653 (!)	0.209 (!)	0.123 (!)	fewer strlen calls (by using sizeof s - 1), writing stdout to /dev/null and with <i>profiling off</i> , but under moderate load; try again (c.23)

86 Jan 24 07:35	0.574	0.216	0.123	as last time, but stdout to tty(!) and under light load. running 15.67 times as fast as B rnews (c.24)
	(!)	(!)	(!)	
86 Aug 8 04:23	0.839	0.51	0.124	performance hit: fflush after each history line for crash-resilience; run for gprof output and calibration with later runs. running under 4.2.1BSD (has 4.3 namei cache) now. real and user times are way up; due to gprof profiling? (c.25)
86 Aug 8 04:24	0.962	0.438	0.131	run with faster ngmatch, with register decls and word- match and STREQN macros; saved 15% user. User time is better than c.25, but still up from c.24. (c.26)
	(!)			
86 Aug 10 07:35	0.805	0.345	0.135	further speedups: ngmatch has more register decls and in-line index; more use of STREQ(N) macro for str(n)cmp in hdrmatch, ngmatch.c and transmit.c; faster ishdr without index. real & user times are better than both c.26 and c.25 (c.27)
	(!)			
86 Aug 11 04:19	1.012	0.303	0.146	rewrote sys.c, used INDEX and STREQ(N) macros throughout rnews. real and sys times are up, but user continues to decline. (c.28)
	(!)			
86 Aug 12 03:51	1.315	0.315	0.154	minor tweaks: all.all.ctl caching, etc. (c.29)
86 Aug 30 17:56	0.564	0.189	0.112	light load, thought we had 3-arg open in fileart, but didn't. Odd. Stopped using gprof. (c.30)
	(!)	(!)		
86 Aug 30 17:57	0.475	0.191	0.095	Really and truly use the 3-arg open. 19 times B rnews speed. (c.31)
	(!)	(!)		

**Data Management:  
A Full-Text Information Retrieval  
Perspective**

Ken Leese  
Director, Technology  
Fulcrum Technologies, Inc.  
560 Rochester Street  
Ottawa, Canada K1S 5K2

**ABSTRACT**

Full-text information retrieval systems have traditionally been applied to a relatively narrow set of applications, and have been directed primarily at information professionals working in somewhat specialized environments.

The range of applications for full-text retrieval systems is now broadening rapidly, both for electronic information distribution and for office systems.

This presentation contrasts the functional and performance requirements of full-text retrieval systems with structured database systems. The largely unstructured nature of textual information and its impact on systems architecture is discussed.

# **UNIX in Health Care: Medical Laboratories - A Case Study**

Sheldon Hamburger  
Management Systems Engineering, Inc.  
32000 Northwestern Hwy, Suite 255  
Farmington Hills, MI 48018

## **1. Introduction - Computers in Health Care**

### **1.1. The Computer as a Diagnostic Tool**

Computers have long been used to assist health care providers in the diagnostic process. The use of computers in sophisticated instrumentation such as CAT scanners is well known. Their tremendous processing power has actually led to the development of specialized devices which would not be possible without this technology.

### **1.2. The Computer as a Research Tool**

Computers are used extensively in the area of medical research. The *information explosion* which has been a byproduct of the computer industry has led to a new need for that information. More and more, medical researchers are turning to computers to provide them with current, accurate, information regarding new procedures and treatments. As newer methods are developed, the results need to be disseminated to the medical world in the most efficient and effective way possible - the computer.

### **1.3. The Computer as an Accounting Tool**

Computers have become a necessity in the area of medical accounting. The increasing complexity of medical accounting procedures brought about by government and insurance regulation can only be accommodated using computer systems. The advent of HMOs has added to this complexity.

### **1.4. The Computer as an Information Tool**

One of the newest applications of computer systems in the area of health care is general information exchange. Nationwide computer networks have been developed to track potential blood and organ donors.

## **2. UNIX in Health Care**

### **2.1. Diagnostic Tool - Health Evaluation Laboratory Program H.E.L.P**

H.E.L.P. is a UNIX based system for use in the medical laboratory. Its development is the subject of this paper. The system is used to automate laboratory testing and preliminary diagnostic services. Laboratory tests are scheduled in the most efficient manner and hard copy schedules are printed. Instruments are interfaced directly to the computer for accurate reporting of test results. These test results are reported back to the physician and maintained in the system for historical reporting and quality control.

### **2.2. Research Tool - Database Software**

Most research software is custom developed. This makes sense since research is generally a *custom effort*. While most research software (as most software in general) has been developed in

conventional programming languages such as COBOL and BASIC, it is increasingly common to see research data in DB based software. The UNIX environment provides some excellent database management packages such as Informix, Unify, Progress, and others.

### **2.3. Accounting Tool – Medical Billing Software**

The accounting area of health care, as with any typical accounting application, has been addressed by many software companies through quite a few commercial packages. The UNIX environment has several medical accounting packages such as MBS (based on Informix-SQL), The Resident (based on Unify), and MDX (written in C).

### **2.4. Information Tool – Communications Software**

Medical information is commonly exchanged between systems via standard communications methods such as 2780/3780 or simple asynchronous transfer. The UNIX world provides an environment for many types of communications. The UNIX operating system provides asynchronous communications with the *cu* utility and synchronous communications with the *uucp* utility. The *uucp* network is the largest computer network in the world.

## **3. UNIX in a Medical Laboratory – A Case Study**

### **3.1. The Need**

The purpose of a medical laboratory is to provide testing and diagnostic services to health care providers. Hospitals have laboratories which they operate for this purpose. Clinics and physicians' offices generally use the services of an independent clinical laboratory. As a matter of fact, some states limit the ability of physicians to operate laboratories.

A medical laboratory processes many requests for services all of which need to be met within a short period of time. Many times, specimens must be collected, scheduled, tested, verified, and reported back to a physician within two or three hours. This is especially true in a hospital environment.

Computers provide the laboratory with the ability to efficiently process large amounts of scheduling information and test results. It is important to note that accuracy and timeliness are of equal importance. A surgeon may require test results while a patient is on the operating table. An inaccurate test result cannot be simply *adjusted* like an incorrect invoice; the catastrophic consequences of errors in this application are indeed life threatening.

### **3.2. The Plan**

The development of H.E.L.P. actually began in 1980. It was at this time that the client began investigating laboratory results systems which were being offered. The high price tags (from \$250,000) and lack of local acceptance made these systems unattractive.

In 1984, MSE, Inc., was contacted about the possibility of custom development of the software for this application. Formal development began in September, 1984. The total system was to include a complete results scheduling and reporting system, quality control, automated results acquisition from on-line instrumentation, and an interface to a billing system.

The system required close to two years to develop. During this time, the laboratory underwent substantial growth requiring the conversion of software and data to several different computer systems – all of them UNIX based. In addition, a new hospital account was added to this independent laboratory and other custom modules were developed which were not foreseen at the initial contract time.

### 3.3. The Result

The system which was developed, H.E.L.P., is a complete software system for medical laboratories. It includes all basic laboratory functions as well as on-line instrument interfacing and a billing interface. The software is currently being used on two Altos 3068 computers with the UNIX System V operating system. The system uses 32 devices including CRTs, printers, modems, multiplexors, and laboratory instruments.

The software is based on the Informix-SQL database system. Instrument interface drivers are written in C. The C shell is used as the UNIX environment. Various UNIX utilities such as *cu* and *uucp* are used. In addition, the Altos network Worknet II is used to network the two computers.

The software was originally written on a Zilog 8000 using UNIX System III. A major new contract signed by the laboratory required moving to two Altos 3068s.

## 4. UNIX Specific Aspects of the Application

### 4.1. The Informix-SQL Database

The H.E.L.P. software is based on the Informix-SQL database system. Informix is the largest selling database system under UNIX. The SQL version, originally released in 1985, conforms to the ANSI level 2 SQL standard. Informix-SQL is a truly relational database and provides many tools for application development. A powerful interface to C is also provided.

This database is portable across many computers, different versions of UNIX (III and V), and Xenix. This is not just a sales claim. The H.E.L.P. system has been successfully ported (in full and in part) to several different computer systems.

### 4.2. UUCP Network

H.E.L.P. can be operated as a true distributed processing system. In the current Altos environment, the Worknet II proprietary network is used to connect the computers on the network. H.E.L.P. also accommodates the *uucp* utility as the basis for the network. As a matter of fact, the original system was developed to run *uucp* and was used for 18 months in that mode. During that time, several different computers were operating on the network.

### 4.3. Real-Time Data Acquisition

The laboratory environment requires that some testing instruments be connected directly to the computer so that test results can be sent electronically to the computer. This improves the speed and accuracy with which results can be entered into the computer.

Today's laboratory testing devices generally provide output on RS-232 lines. This makes hardware interfacing relatively simple. Unfortunately, this output is not always ASCII data. In addition, the protocol (number of data and stop bits) varies with each instrument.

We wrote a general C language instrument driver which has been used successfully with many different instruments. Much of the initial development of interfacing methods involved the use of the *cu* utility to capture data. While convenient for developmental purposes, it was not adequate for use on a regular basis.

### 4.4. The C Shell

We have chosen the C shell as the operating environment because of its versatility and convenience for program development.

#### **4.5. Programming Standards**

The UNIX operating system is an excellent programming environment. In order to take full advantage of its facilities and features, we have devised programming standards which are used in all company developed software. The use of these standards produces easily maintainable and portable code.

Programming standards include C shell scripts, SQL queries, and Informix modules. In addition, script names, file names, and database field names are also chosen using company standard rules.

#### **4.6. Portability**

The UNIX operating system by nature provides portability between computers. This does not guarantee portability of application code. But careful planning can give a reasonable guarantee of portability.

The use of a portable database such as Informix-SQL is a good start toward portability. In addition to selecting Informix as the database for our applications, C programs which access the database also use the Informix libraries which make these programs portable.

The C language is by nature portable. However, certain aspects of C are machine dependent and should be considered when programming in that language. We consider these areas in the development stages and have never had to change code in order to compile on different machines.

### **5. Conclusion**

#### **5.1. UNIX Provides a Reliable Multi-User Environment**

Today's applications call for true multi-user operation with full concurrency control. UNIX provides multi-tasking and newer kernels provide multi-user locks.

#### **5.2. UNIX Provides a Powerful, Stable Development Environment**

In order to develop quality applications within reasonable budgets, computer systems with strong development environments are needed. UNIX provides a versatile shell environment and plenty of utilities for programmers.

#### **5.3. UNIX Provides Software Portability**

As users' requirements change and their businesses grow, the need for larger systems becomes apparent. UNIX provides for portability of software and data allowing users to keep their largest investment as they move.

---

Sheldon Hamburger is president and founder of Management Systems Engineering, Inc. After founding the company in 1978, Mr. Hamburger directed his company's efforts toward the production of reliable and portable multi-user business software. His company specializes in health care applications.

Mr. Hamburger received his Bachelors degree in computer engineering from the University of Michigan and is founder of michigan!usr/group, the Michigan affiliate of /usr/group.

# RDBMS Features and Data Integrity Issues in an Army Budget Database System

Patricia J. Ton  
IIT Research Institute  
Rome, NY

December 1986

## Abstract

This paper describes the Budget Database System and highlights the application of certain features of a relational database management system in achieving system design goals. It also relates how data integrity problems were addressed on individual and multiple user site configurations.

## 1. THE BUDGET DATABASE SYSTEM

Since 1982, IIT Research Institute (IITRI) has developed two database management systems for the U.S. Army, which assist managers in various aspects of the budgeting and planning process. In this paper, they will both be referred to as the Budget Database System.

The Budget Database System is operational at 16 different Army sites across the country, including Army (AMC) Headquarters, and is used to assist managers in the following areas:

- o Ranking projects for funding purposes
- o Monitoring how shortcomings are being addressed
- o Project justification
- o Project management and cost controls
- o Project planning and execution

Data is entered and analyzed at the individual user sites and is merged an average of three times a year at headquarters (see Figure 1). The purpose of the data merge at headquarters is to give an overall ranking of the R&D projects from the individual user sites. The ranking determines which projects will obtain funding in the coming years.

The Budget Database System comprises a combination of UNIX<sup>1</sup> Shell programs, C, and the INFORMIX<sup>2</sup> relational database management system (RDBMS). It runs under UNIX System V and INFORMIX Version 3.3. The makefile and Source Code Control System (SCCS) tools are also utilized for configuration control. The user interface is a menu system, also developed at IITRI, which enables one to input funding data (at various levels of detail) into the budget database and obtain up to 40 management reports.

<sup>1</sup>UNIX is a trademark of AT&T Bell Laboratories

<sup>2</sup>Informix is a registered trademark of Informix Software, Inc.

This paper describes how the features of the Informix RDBMS were used to effectuate our system design goals and the steps that were taken to address data integrity problems.

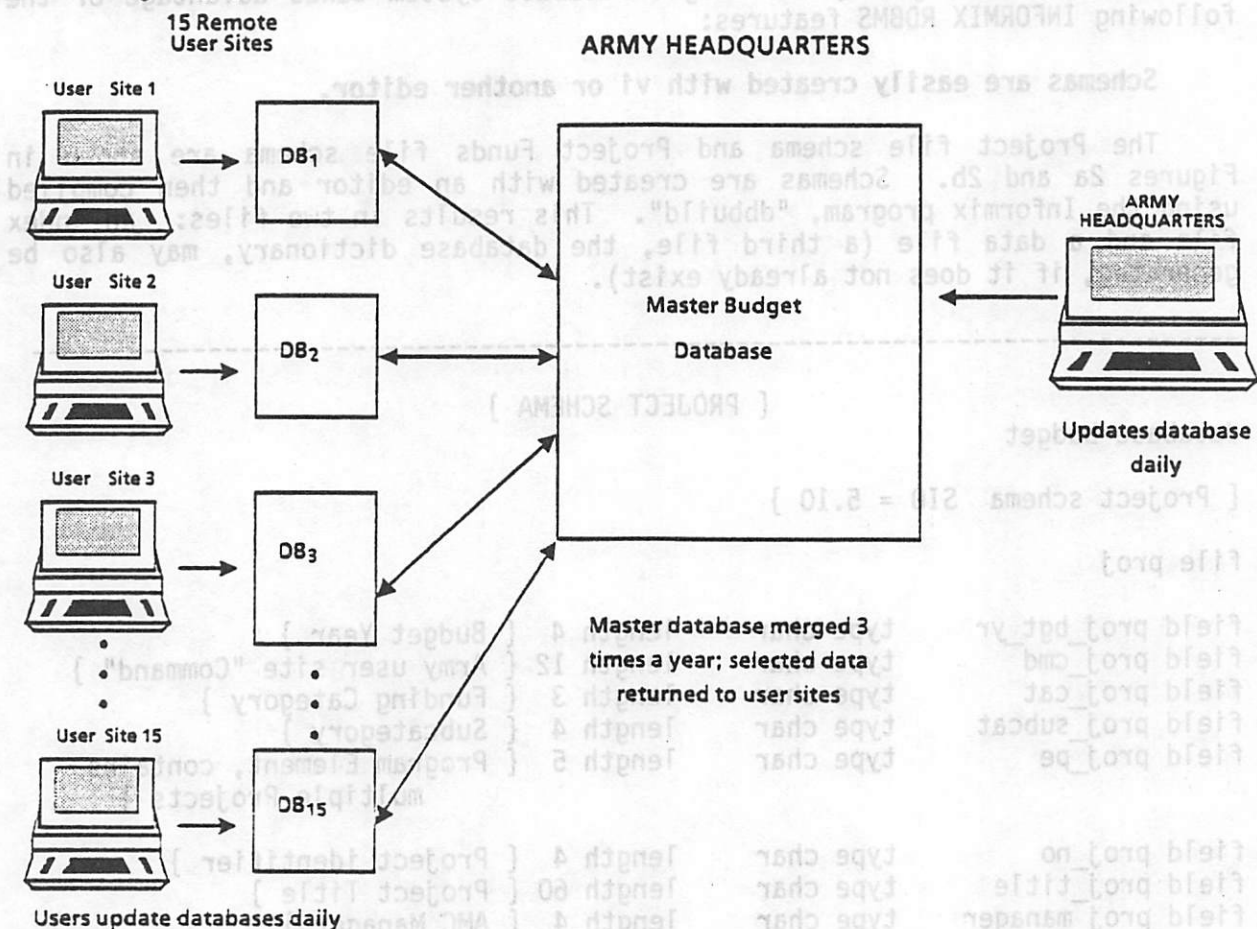


Figure 1. The Data Merge Process

## 2. SYSTEM DESIGN GOALS

Our first goal was to model the structure of the customer's data and business practices.

Our second goal was to write flexible, maintainable code that would allow changes on demand and functional expansion without resulting in "spaghetti code". This was important because changes in management often led to changes in budgeting procedures.

Because many of the users are budget analysts and data entry personnel, and are not computer programmers, our third goal was to present a friendly, understandable user interface so that the automation of the budget and procurement process would be a natural transition for the user. This was especially important in the data entry area, where the majority of manhours would be spent.

### 3. INFORMIX RDBMS FEATURES

To meet these goals, the Budget Database System takes advantage of the following INFORMIX RDBMS features:

Schemas are easily created with vi or another editor.

The Project file schema and Project Funds file schema are shown in Figures 2a and 2b. Schemas are created with an editor and then compiled using the Informix program, "dbbuild". This results in two files: an index file and a data file (a third file, the database dictionary, may also be generated, if it does not already exist).

---

```
                                [ PROJECT SCHEMA ]
database budget
[ Project schema  SID = 5.10 ]
file proj
field proj_bgt_yr      type char      length 4 { Budget Year }
field proj_cmd         type char      length 12 { Army user site "Command" }
field proj_cat         type char      length 3  { Funding Category }
field proj_subcat      type char      length 4  { Subcategory }
field proj_pe          type char      length 5  { Program Element, contains
                                                multiple Projects }

field proj_no          type char      length 4  { Project identifier }
field proj_title       type char      length 60 { Project Title }
field proj_manager     type char      length 4  { AMC Manager }

field proj_idx         type composite   { Index used in data entry }
                        proj_bgt_yr,    { screens and reports }
                        proj_cmd,
                        proj_cat,
                        proj_subcat,
                        proj_pe,
                        proj_no          index primary

permissions
file      group others      access read
file      group budget      access read update add delete
file      user dba          access all    { includes schema control }

end
```

---

Figure 2a. Project Schema File

```

-----
{ PROJECT FUNDS SCHEMA }

database budget
file projfund
field projf_bgt_yr      type char      length 4    { Budget Year }
field projf_cmd         type char      length 12   { Command }
field projf_cat         type char      length 3    { Category }
field projf_subcat      type char      length 4    { Subcategory }
field projf_pe          type char      length 5    { Program Element }
field projf_proj        type char      length 4    { Project }

field projf_yr          type integer    { Project Fund year }
field projf_guid        type long       { Allocated funding }
field projf_fund        type long       { Amount funded }
field projf_unfund      type long       { Amount unfunded }

field projf_proj_idx    type composite  projf_bgt_yr,
                                     projf_cmd,
                                     projf_cat,
                                     projf_subcat,
                                     projf_pe,
                                     projf_proj      index dups

field projf_idx          type composite  projf_bgt_yr,
                                     pprojf_cmd,
                                     pprojf_cat,
                                     pprojf_subcat,
                                     pprojf_pe,
                                     pprojf_proj,
                                     p projf_yr      index primary

permissions
file      group others    access read
file      group budget    access read
file      user dba        access all

end
-----

```

Figure 2b. Project Funds Schema File

Schema changes are made simply by editing the schema source and recompiling; no unloading of the data is necessary.

The length of time it takes to recompile depends on how much data there is in the database. Field lengths, name changes, indexes, or field types can also be changed using the same procedure. Dbbuild shows a list of the changes made and requests confirmation that the change should be made. The dbbuild utility makes it relatively easy to add/modify fields/indexes when there are urgent requests for schema changes.

**File and field permissions are used to restrict selected users and groups.**

Some user sites had a need to restrict users from funding and priority fields, and designated a privileged user, "dba" (the "Database Administrator"), to enter the funding and to take responsibility for general database administration.

The permissions section of the schema is specified after the database, file and fields are declared. Permissions can be specified on a file or field level and can differ in each file in the database. Additionally, a specified user or group of users can have certain levels of access (i.e., add, delete, update, control, none, or all). As seen in the Project schema, the permissions feature allows one person or group of designated personnel the ability to access the funding fields, while allowing others to modify Project related information other than funding.

The "dba" user has total file privileges, including "control" of the database files, which means that dba is the only user able to restructure the files. Users in the group "budget" are able to have all privileges except "control", and users in group "others" can read but not alter the data in the Project file. The Project funding file has restrictions on who is allowed to modify funding. Thus, in this file, "budget" and "others" have read-only privileges.

**Data entry screens are custom designed and easily created using vi.**

As with the schemas, data entry screens (or PERFORM<sup>3</sup> screens, as they are called) are created with vi. The screen source code is compiled using the INFORMIX program, "formbuild," and results in a file with the suffix, ".frm". Changes to the screen are made in the source file with vi and recompiled in the same way.

However, data entry screens do not have to be customized. A quick data entry screen can be generated using the "formbuild -d" command. This can also be used as a template for a customized data entry screen.

**The three sections of the Perform screen provide a friendly data entry interface, program flexibility, and data control.**

The three sections of a PERFORM screen are: the screen section, the attributes section, and the instructions section. The screen section (refer to Figure 3a) designates the format of the data entry screen as seen by the user. The attributes section maps the fields on the screen to the data elements in the schema file and is also a vehicle for executing other field related instructions. The instructions section is for more detailed control of fields, files and operations.

The next sections highlight some of PERFORM's features that have been integrated to promote user-friendliness and to ensure data integrity.

---

<sup>3</sup>Perform Custom Screen Generation and Transaction Processing Package, a trademark of Informix Software, Inc.

## Screen Section.

The data entry screens are the user's interface to the database, serving as a means to add, modify, delete and query information in the Budget Database System. If a data entry screen is generated using "formbuild -d", the fields will be listed in the order they are specified in the schema, with the actual schema names (mnemonic or otherwise) beside them as shown below.

```
database budget
screen
{
proj_bgt_yr      [f000]
proj_cmd         [f001]
proj_cat         [a0]
proj_subcat      [f002]
proj_pe         [f003]
proj_no         [f004]
proj_title       [f005]
proj_title       [f006]
proj_manager     [f007]
}
end
attributes
f000 = proj_bgt_yr;
f001 = proj_cmd;
a0 = proj_cat;
f002 = proj_subcat;
f003 = proj_pe;
f004 = proj_no;
f005 = proj_title[1,30];
f006 = proj_title[31,60];
f007 = proj_manager;
end
```

The data entry screens for the Budget Database System, however, are customized in an attempt to group data elements in a logical and familiar way for the user (see Figure 3a). Most of the data entry screens contain data elements from one or more schema files.

In addition, the schema names of the fields are replaced by descriptive names that the user is familiar with (e.g., Command, Program Element, AMC Manager). Attribute names are located within the delimiters ("[]") and are linked to the schema names in the attributes section. The "-d" option provides arbitrary attribute names (f000, f001, a0, etc.). In the Budget Database System, attribute names are mnemonically renamed (e.g., cmd, proj, title) to assist the programmer when changes have to be made to that particular data field.

-----  
database budget

screen

{

-----[b\_yr]-----

PROJECT DATA ENTRY SCREEN

Command : [cmd ]

Program Element : [pe ]

Category : [cat][scat]

Project : [proj]

=====

TITLE : [title ]

AMC Manager : [m]

Fiscal Year	AMC Guidance	Funded Amount	Unfunded Amount	Project Adjustment
=====	=====	=====	=====	=====
[year]	[guid ]	[fund ]	[unfund ]	[adjust ]

(Press 2F to display the Project funding)

(Funded and Unfunded fields are for display only, and cannot be changed.)

=====

}

end

-----  
Figure 3a. Screen Section of the Project Data Entry Source File

-----  
Other design features used in the screen section:

- o Certain key fields are set apart at the top of the screen to distinguish them from the rest of the data elements
- o Helpful messages in parentheses are included at the bottom of the data entry screens, informing the user how to access additional screens.

### Attributes Section.

The second section of a PERFORM screen source file is the attributes section (see Figure 3b). Within the attributes section, operations can be performed on individual fields.

---

#### attributes

```
b_yr = proj_bgt_yr = projf_bgt_yr, REVERSE, NOUPDATE,  
      DEFAULT = today "yyyy", include = (1700 to 2100);
```

```
cmd = proj_cmd = projf_cmd,  
      UPSHIFT, reverse, REQUIRED, noupdate;
```

```
cat = proj_cat = projf_cat,  
      upshift, reverse, required, noupdate,  
      COMMENTS = "(Valid answers: 6.1, 6.2, 6.3, 6.4, 6.5, 6.7)",  
      INCLUDE = ("6.1", "6.2", "6.3", "6.4", "6.5", "6.7"),  
      AUTONEXT;
```

```
scat = proj_subcat = projf_subcat, upshift, reverse, noupdate;
```

```
pe = proj_pe = projf_pe, reverse, required, noupdate;
```

```
proj = proj_no = projf_proj, upshift,  
       reverse, required, noupdate;
```

```
title = proj_title, upshift;  
m = proj_manager, upshift;
```

```
year = projf_yr, include = (1970 to 2099), noupdate;  
guid = projf_guid, NOENTRY, NOUPDATE, RIGHT;  
fund = projf_fund, noentry, noupdate, right;  
unfund = projf_unfund, noentry, noupdate, right;  
adjust = DISPLAYONLY type long, reverse, right;
```

---

**Figure 3b. Attributes Section of the Project Data Entry Source File**

---

The "reverse" command highlights important fields in the data entry screen with reverse video. Key sorting fields in the reports, display-only fields, and special entry items are highlighted in this manner to distinguish them from other fields.

The "default" command is used with fields such as budget year. This feature automatically sets the budget year value to the current calendar year but allows the user to input other values, if desired.

The "comments" command provides additional explanation concerning the required input values of a particular data element. The comments text appears when the user's cursor reaches that particular field on the screen. The comments command is used with the "subcat" field.

The "include" command forcibly restricts the user to certain values. In the example, the user must input one of the following values to the subcat field: 6.1, 6.2, 6.3, 6.4, 6.5, or 6.7. This is one of the features used for maintaining data integrity.

The "upshift" command translates characters to upper case. This command helps standardize user responses. For example, one user may input "Avscom" to the Command field while another may enter "AVSCOM". Certain fields are automatically upper cased for data consistency. This also saves the reports from having to check data for various cases based on whether a value was entered in upper case, lower case, or both.

The "right" command right justifies data elements. Fields such as the funding fields are right justified, for data consistency.

The "required," "noenter," and "noupdate" commands assist in maintaining data integrity. The reports require certain fields to be filled in for correct results. The "required" command helps ensure this when a user adds a new record. A new record cannot be added if a required field is not filled in. Conversely, there are times when the user's input is not desired, as in the case of the Project funding fields. The values in these fields cannot be altered, but they can be reviewed. The "noenter" and "noupdate" commands are used on the "guid", "fund" and "unfund" attributes.

The data entry screen also contains a "display-only" field whose attribute name is "adjust". The Project Adjustment field does not exist in the schema; it is placed on the screen for informational purposes only. The adjust field is the difference between the Funded Amount and AMC Guidance. Because it is a display field, it is highlighted to stand out from the other fields. We will elaborate on the display-only field in the next section.

### Instructions Section.

The instructions section (see Figure 3c) is used to provide additional user convenience and data control. The example in Figure 3c demonstrates:

- o A way to prevent orphan records
- o An expedient record search feature
- o Autocalculation of a display-only field.

```
-----
instructions
composite join *proj_idx = projf_proj_idx;
proj master of projfund;

after query display of proj projfund
    let adjust = fund - guid

end
-----
```

Figure 3c. Instructions Section of the Project Data Entry Source File

Orphan records can result if a user deletes a parent record but forgets to delete its repeating group records. For example, suppose a Project record exists with eight years of Project Fund records. There would be eight "orphan records" if the user deletes the Project record but not the corresponding Project Funds. Similarly, if a user adds a Project Funds record but forgets to add the Project record, an orphan record would result.

The first line under the instructions section is a composite join on two indexes found in the Project and Project Funds schemas (refer also to Figure 2). Both indexes are made up of the following logical key fields:

Budget Year  
Command  
Category  
Subcategory  
Program Element  
Project

Recall that these fields were also highlighted at the top of the data entry screen.

The asterisk before the proj\_idx field means that if the user is adding a Project Funds record, the values in projf\_proj\_idx must already exist in a Project record. Thus a Project Funds record cannot be added without first adding a Project record having the same key fields. Also, the user cannot delete a Project record without first deleting the Project Funds records associated with it (i.e., having the same key fields). This keeps the number of orphan records to a minimum.

Although fields from both the Project and Project Funds files are shown on the data entry screen, only the fields from one file can be accessed at a time. To switch between the two files, the user types "f". The current file's name will appear in the upper right of the screen and the "[ ]" delimiters will appear adjacent to that file's data elements. The user can then proceed to query, add or modify the data in the active file, given the proper permissions.

When the user changes between the Project and Project Funds files, the composite join will retain the values in the key fields. This saves time when adding or querying records. For example, after a Project record is added, the Database Administrator may want to add a Project Funds record. Instead of having to retype the key fields, the composite join command retains them. Thus, the user has only to input the fiscal year and funding information. The key fields will also be retained if the user changes files and performs a query.

Another method of querying records is to use PERFORM's "master" and "detail" commands. Note the second instruction line in Figure 3c. The "master of" instruction automatically searches the associated Project funding records based on the key fields retained by the composite join. As an example, assume the user queries a Project file. If a "d" (for "detail") were typed, all the Project funds records whose key fields matched those of the Project's are automatically queried. If the user wants to return to the "master" Project file, an "m" is typed.

For the user's information, a display-only field has been added. Observe the third instruction in Figure 3c. This instruction automatically calculates the Project Adjustment field when Project or Project Funds records are queried/displayed. This field does not appear in either schema, and thus is not entered or modified; it is for informational purposes only. Autocalculations and automatic data entry can be made on normal data elements as well, and can ease the burden of entering data.

### 3.3. Ace<sup>4</sup> Report Generator and C Interface Tool

The Ace report generator is used for quick generation of reports. It is very easy to use and one can generate working programs quickly. Ace was used in response to "last minute" report requirements.

However, there are some limitations with Ace. One limitation is that only 100 variables and parameters are allowed per Ace report. Some of the budget reports contain subtotals, totals and adjustments which require more than 100 variables to be kept.

Also, the speed at which an Ace report executes is related to the amount of data processed. Most of the budget reports process thousands of records. Thus, reports which process large amounts of data or whose output is time critical are written using the INFORMIX application language library (ALL-II)<sup>5</sup> interface and the C programming language. ALL-II allows the programmer to access and modify the database information using C. The majority of the reports in the Budget Database System are written with ALL-II and C.

## 4. SOLVING DATA INTEGRITY PROBLEMS AT INDIVIDUAL SITES

In the Budget Database System, the philosophy is to try to catch data integrity errors where they originate--with data entry. Most data entry errors are intercepted using the inherent features of the PERFORM screens, some of which are described above. However, additional data integrity tools are incorporated into the Budget Database System with the rationale that if data is correct at the individual sites, the data would retain its integrity when it is merged at headquarters. The additional data integrity tools are described below:

- o The structure of the database is such that most records have parent or child relationships. Validation programs were written which traverse the structure of the database and identify orphan records.
- o Many data fields expect a certain range of values. If the user bypasses the data checks built into the data entry screens, there are programs which inspect critical fields. Invalid fields are brought to the user's attention.

<sup>4</sup>Ace Relational Report Writer, a trademark of Informix Software, Inc.

<sup>5</sup>ALL-II application language library routines for the C Programming Language

- o There are also programs that automatically calculate certain fields based on previously entered items and algorithms, thus bypassing user input completely.
- o A number of the C reports check for data integrity errors internally and print out useful error messages. Some messages reference the primary key of the delinquent record or a specific program is named which can be executed to correct the error.

The purpose of the validation routines is to ensure that the user obtained correct results when running the management reports. There are usually two scenarios a user goes through before obtaining a report: (1) the user makes one or two changes before obtaining a report, or (2) the user adds/modifies a number of records before generating reports. Recognizing this, the Budget Database System offers the user two convenient places to run the data validation programs:

- (1) After exiting a data entry screen containing the fields to be validated, or
- (2) From the System Administration menu.

## 5. SOLVING DATA INTEGRITY PROBLEMS - MULTIPLE SITE CONFIGURATION

The user sites' data are merged in a master database approximately three times a year (see Figure 1). To make the process as easy as possible for the users, the entire procedure was automated. At the individual user sites, data is selectively unloaded from the database and is mailed to headquarters on tape. At headquarters, computer operators run the data through a merge process using a Bourne shell menu interface. After the data merge, data tapes are created and returned to the individual user sites. The tapes contain selected files from the merge and are used in updating the user sites' databases.

Most files in the master database are automatically updated by the corresponding incoming files according to the user site responsible for the data. Responsibility for the data is partitioned among the user sites prior to the merge.

A database file (i.e., a control file) is created to map the sites to the data for which they are responsible. The responsible user site's records override any others in a conflicting situation. The control file becomes a part of the database only during the merge, so that if responsibilities change, the control file can be updated accordingly.

If a conflict arises which cannot be resolved through the control file method, the record in question is written to a temporary file whose format is identical to the master file's format with the exception of an additional update flag field. A functional decision maker then scans the records in conflict, which are presented by means of a data entry screen. By changing the value of the update flag from "D" (for Delete) to "R" (for Replace), the functional decision maker indicates which records are to replace the

corresponding master record having the same primary key. The records in the temporary file default to being deleted.

After completion of the data merge, the same validation programs which were run at the individual sites are run on the merged data in the master database. Any errors found are corrected before the data tapes are released to the user sites.

## 6. SUMMARY

With all its data entry checks and validation programs, orphan records and invalid data still appear. For the users who insist on "doing it their own way," there is little recourse. Although users are discouraged from modifying the database structure, some users modified the schemas in an attempt to further customize the system. Unfortunately, when those sites submitted their data to be merged, it was in the wrong format. However, for the usual data integrity problems, the tools exist to find and correct inconsistent links and other "bad data".

Generally, the Budget Database System has been praised by its users. Over the past four years, the number of user sites IITRI directly supports has grown from 10 to 16. In addition, the Army has expressed its appreciation for IITRI's ability to make changes on demand. The UNIX operating system, INFORMIX 3.3 RDBMS, Make, and SCCS have contributed to providing a productive and efficient programming environment.

## 7. Acknowledgements

I would like to acknowledge the other members of the "Wolfpack" team who created, tested, debugged and documented the Budget Database System (a.k.a. the RDA/MAMP System): Mark Costello, Frank Miccoli, Kimberly Free, Michael Piacentino, Amy Vrabel, Ralf Durkee, and Vinny Cavo. Our fearless leader is Wolfhart Goethert.

Also, "good luck" to Gary Peck and Donna Wagster, and personal thanks to Ms. Donna Crossland for her time and effort to produce this paper.

## References:

1. RDA/MAMP User's Manual, Release 6.1, IIT Research Institute, October 1986
2. RDA/MAMP Programmer's Manual, Release 6.1, IIT Research Institute, October 1986
3. INFORMIX Relational Database Management System User's Manual, Relational Database Systems, Inc., November 1984

The Johns Hopkins Hospital has been developing and operating UNIX-based DBMS applications for on-line transaction processing during the last two years. These systems operate in a program distributed environment of UNIX, IBM MVS IMS and MUMPS systems. Universal access to common information is provided by use of remote procedure call methods and intelligent gateways. Reliable multi-site updates are handled by a centralized transaction distribution facility under UNIX. Problems of centralized and distributed data administration, applications demands, workloads, product limitations and UNIX limitations are discussed. The distributed model, some of the applications profiles, hardware and software selections and future directions are reviewed.

## 1. Introduction

In early 1984 the Johns Hopkins Hospital (JHH) began several projects to develop new clinical information systems and to integrate diverse existing systems. The JHH operates under decentralized administration which has fostered the development of a heterogeneous collection of independently managed systems. Some low-speed, ad-hoc links existed to provide information transfers for important applications. However, the concept of common, centralized data shared by applications on different machines, and the concept of distributed data autonomously maintained by departments yet available for remote access did not exist. A plan was formulated to enable centralized and distributed data administration and effective networking of systems to support process-to-process communications for remote access. The plan called for the following:

- ☐ a networked database "machine" for multiple system access to common patient data
- ☐ use of relational database management for the central database as well as for the local databases supporting new applications
- ☐ use of the UNIX operating system for development and production to assure reasonable portability of applications to new hardware over the course of time
- ☐ use of local area network technology, specifically Ethernet, and development of gateways and protocols to enable networking of non-UNIX systems in existence
- ☐ adoption and implementation of a model for distributed computing across very heterogeneous hardware and software
- ☐ implementation of methods to "commit" updates across several heterogeneous sites in a simple, practical, reliable (but not necessarily generally correct) way; concern for not losing updates to out-of-service systems dominated concern for globally correct serialization
- ☐ simple to use, intuitive forms-based user interfaces for terminal oriented applications and window-based applications for workstation applications.

The responsibility for developing and implementing this plan was given to the newly formed Operational and Clinical Systems Division (OCS) of the Hospital. The division started working on the implementation in mid-1984.

# Real World UNIX DBMS Applications: Experiences and Observations

Stephen Tolchin †, Eric Bergan †† and Marvin Schneider ‡

The Johns Hopkins Hospital  
(† and The Applied Physics Laboratory)  
(†† and xyzy, Inc.)  
(‡ and SOFTA Technologies, Inc.)

## ABSTRACT

The Johns Hopkins Hospital has been developing and operating UNIX-based DBMS applications for on-line transaction processing during the last two years. These systems operate in a pragmatic distributed environment of UNIX, IBM MVS IMS and MUMPS systems. Universal access to common information is provided by use of remote procedure call methods and intelligent gateways. Reliable multi-site updates are handled by a centralized transaction distribution facility under UNIX. Problems of centralized and distributed data administration, applications demands, workarounds, product limitations and UNIX limitations are discussed. The distributed model, some of the applications profiles, hardware and software selections and future directions are reviewed.

### 1. Introduction

In early 1984 the Johns Hopkins Hospital (JHH) began several projects to develop new clinical information systems and to integrate diverse existing systems. The JHH operates under decentralized administration which has fostered the development of a heterogeneous collection of independently managed systems. Some low-speed, ad-hoc links existed to provide information transfers for important applications. However, the concept of common, centralized data shared by applications on different machines, and the concept of distributed data autonomously maintained by departments yet available for remote access did not exist. A plan was formulated to enable centralized and distributed data administration and effective networking of systems to support process-to-process communications for remote access. The plan called for the following:

- ☐ a networked database "machine" for multiple system access to common patient data
- ☐ use of relational database management for the central databases as well as for the local databases supporting new applications
- ☐ use of the UNIX operating system for development and production to assure reasonable portability of applications to new hardware over the course of time
- ☐ use of local area network technology, specifically Ethernet, and development of gateways and protocols to enable networking of non-UNIX systems in existence
- ☐ adoption and implementation of a model for distributed computing across very heterogeneous hardware and software
- ☐ implementation of methods to "commit" updates across several heterogeneous sites, in a simple, practical, reliable (but not necessarily generally correct) way; concern for not losing updates to out-of-service systems dominated concern for globally correct serialization
- ☐ simple to use, intuitive forms-based user interfaces for terminal oriented applications and window-based applications for workstation applications.

The responsibility for developing and implementing this plan was given to the newly-formed Operational and Clinical Systems Division (OCS) of the Hospital. The division started working on the implementation in mid-1984.

The databases which have been built are neither small nor simple. The central patient identification relation on the database "machine" has tuples for about 2.5 million patients, over 1% of the U.S. population. This is but one relation in a database of 45 relations with complex data semantics. Data in this central database must be maintained with reliable links to departmental databases on other systems.

In addition to complexity and size, the data access patterns require a moderately high volume of transaction processing in a highly concurrent mix of reading and writing transactions. For some applications now under development, 20,000 update transactions per hour must contend with many concurrent readers who need to access large regions of the database per read. Data integrity is critical since clinical information is involved. Furthermore, many of the systems need to run 24 hours per day, 7 days a week so high availability must be assured. The load, consisting of programs on local and remote systems as well as terminal and workstation users, is expected to grow to several hundred, perhaps a thousand, simultaneous users as applications proliferate throughout the 150 clinics and 50 nursing units of the institution.

## 2. Environment and Systems

The systems which existed in early 1984 and which still exist today are:

- ☐ three PDP 11/70's running the InterSystems MUMPS operating system in native mode which are used by the Department of Laboratory Medicine to process Lab orders, enter results and report results. These systems also support Surgical Pathology, Microbiology, Blood Bank and other applications. The three systems are connected by high speed interfaces and can share data across them using InterSystems DDP distributed data access protocol. MUMPS *does not support any industry standard communications protocols* so there is no way to run IP/TCP or XNS. Furthermore, MUMPS does not allow users to develop new drivers for this operating system. The only data type supported by MUMPS is ASCII string. However, these systems have excellent price / performance.
- ☐ two PDP 11/70's running the same MUMPS operating system are used by the Oncology Center to run a very sophisticated clinical management system [8] for in- and out-patients. The inpatients, however, are admitted through an IBM MVS 3081 based system.
- ☐ a PDP 11/84 run by the Anesthesiology Department which schedules patients, physicians and operating rooms as well as supporting other Department needs. This is also a MUMPS system.
- ☐ two PDP 11/70's, soon to become one VAX 8650, are operated by the School of Medicine and are used for professional fee billing. The VAX will run VMS with InterSystems M/VX MUMPS product hosted under VMS. It is possible for MUMPS and VMS to communicate, thus we have hopes of sensible TCP/Ethernet communications with this system.
- ☐ large mainframes — an IBM 3081 and 3083 (soon to become a 3090/200) running MVS and VM (production all under MVS), CICS, VTAM. These systems are operated by the Information Systems Division and do all financial and most administrative data processing for the institution. They also operate an inpatient Pharmacy system and will soon run a new, very large admitting, discharge, transfer and lots more system. These latter two software systems are the responsibility of OCS. They are developed using PCS/ADS, an IBM product which uses IMS for database support. The mainframes had run a patient identification system (PID) as a heavily indexed VSAM file of 1.6 million patients. Also, the IBM runs a Radiology reporting and MIS system, which is being replaced by a new UNIX-based system. Inpatient nursing units are equipped with 3278s for access to limited duration Lab and Radiology results; the Lab data is uploaded from the PDP's via a 4800 baud link. We are replacing this with a much more extensive system using Sun workstations on each nursing unit.
- ☐ a VAX 11/750 which OCS operated but did not own was used to run an Emergency Medicine system under VMS - M/VX.
- ☐ other systems exist but are of less interest. However, countless PCs and their users hunger for access to clinical and administrative (MIS) data. Workstations and MicroVaxes seem to appear on a weekly basis.

The challenges we faced were: (a) a lack of applications and a tremendous backlog of needed applications development, (b) a need to develop integrated solutions via networking and distributed processing across these environments, (c) the need to select a basis for new development including operating system, hardware, DBMS, distributed computing and network technology.

We had recently completed a large, distributed OA system for the Army. This had given us some useful experience and an opportunity to evaluate several UNIX based, or related DBMSs.

### 3. Decisions

The first step in implementing these plans was to choose the hardware and software to be used for new applications systems, for integration of existing systems and for development and production. From the beginning, we decided to use UNIX as the cornerstone for our development. This was for the usual reasons — hardware vendor independence, ease of development, and the necessary foundations existed in UNIX to consider building a production distributed system. Further, there appeared to be several relational database systems from which to choose, running under UNIX on a variety of hardware products.

#### 3.1 Database Management System Decision

We needed to select a DBMS that ran under UNIX and preferably other operating systems as well. Our requirements as a development / DP organization were quite different from those of an OEM / software re-seller. We required:

- full journalling and recovery capabilities
- atomic commitment of multi-statement transactions
- good "front-end" tools for development of user applications (although we were wary of, and avoided 4th generation tools which might not be powerful enough for applications maintenance and which might negatively impact performance)
- a clean interface between the data management language (i.e. SQL or QUEL) and C
- vendor commitment to improving performance
- ability to scale up the hardware easily and under our control
- if we could have gotten it, we would have liked enforcement of integrity rules and domain rules.

In early — mid 1984, the closest system to meeting these requirements was Ingres from Relational Technology, Inc. We had evaluated version 2.1 earlier and were impressed with the enhancements to be released in version 3.0, which met many of our needs. Since performance was a concern, especially for the centralized component of our architecture, we seriously considered the Britton-Lee IDM-500 as well. We had recently acquired an IDM for the Army project and had evaluated it as well as Ingres and Unify, and, to a somewhat lesser extent, Informix. While Oracle did not yet run under UNIX, we obtained comparative evaluations for VMS of Oracle, Ingres and the IDM which were done by TRW for the Space Telescope Science Institute as well as Navy user reports. We also ran the "DeWitt" [9] benchmark suite and obtained multi-user benchmarks from one of the national laboratories. We spent considerable time speaking with Britton-Lee users and talking with company representatives. While we believed the multi-user performance of the IDM exceeded the software-based products of the time, we had several concerns:

- there were no company supported high-level screen and reports development tools
- the portable IDEL libraries weren't completely implemented yet
- use of specialized hardware implied a unique, non-generally re-usable architectural element; it was possible that we could duplicate the performance by using a dedicated supermini and software DBMS as a network server, and that we could easily move the software to faster machines whenever necessary
- we had concerns about field support in a manner timely enough for our needs in the Hospital

- we needed to support databases on local machines as well, and using a single software product made sense
- The only communications options were either RS-232, which was too slow for our needs, or Ethernet running the XNS protocol suite, which was and remains less important and common than TCP/IP.

At the time of our investigations the current industry ferver about SQL did not exist, so we preferred and felt comfortable with QUEL as implemented by Relational Technology/Ingres or Britton-Lee/IDL. We decided to use Relational Technology Ingres as our DBMS and as our forms-based user interface development package. We began under VMS which enabled us to move quickly to Ingres 3.0, and we switched development to a Pyramid 90x UNIX system (under 4.2BSD) using a beta version of Ingres 3.0 in the Spring of 1985. We continued development and beta testing through releases 4.0 and 5.0. We have several production systems now running on Pyramid 98x systems. Our architecture, which unconventionally splits front and back ends into client and server pieces for performance and maintenance reasons, is described in Section 4.3.

We continue to investigate new DBMS systems and have arranged to be a beta site for the Sybase DataServer/DataWorkbench products on Sun 3. This product represents a new generation in UNIX-based database products, with a significantly different architecture than has been used by previous database systems (but particularly attractive to us since the front-end, back-end split accommodates our existing architecture.) Some of the advantages we see of this product over existing relational DBMS products include:

- performance appropriate for high volume on-line transaction oriented systems which need to support a large number of users in concurrent read/write usage (vs. existing software systems which are architected primarily for low transaction volumes and decision support applications)
- conservation of expensive memory resources and efficient use of shareable memory for performance objectives by using a multi-threaded server-based architecture which implements its own operating system functions and appears to UNIX as a single process (thus also avoiding expensive task switching and expensive system calls for inter-process synchronization as well as enabling more efficient scheduling). Memory is used for the single database process and for caching indices and caching compiled procedures, rather than for supporting unnecessary duplication of large back-ends for each user, which is typically required by other systems. While some systems such as Ingres support a form of compiled query (a "repeat"), these are not shareable across processes resulting in considerable memory consumption. The "repeat" can also lead to inconsistent query plans across different invocations of the application
- rectification of UNIX deficiencies by implementing non-blocking disk I/O and by avoiding the UNIX file system and associated overhead
- support of domain rules and defaults within the database, as well as support of SQL triggers within control-flow extended SQL; thus enforcement of integrity is entirely within the database and not in the front-end applications. This significantly reduces front-end program complexity and maintenance. Also, the database becomes secure in network environments from program access by front-end applications. Since not all applications development is within our control, such accesses may cause integrity violations. This is important for us since we wish to provide database access to other development shops. Handling integrity within the database also improves performance by reducing the number of accesses into the database by the front-ends
- support of high availability by enabling dumps of the database during live transaction processing, and assuring that the dumps represent an instantaneous point of time for recovery purposes. This provides non-stop operations of the software, which is critical in the Hospital
- support of maintenance coupled with availability by allowing schema changes during live transaction processing
- the UNIX file system is bypassed: relations and/or databases are not stored as UNIX files for performance reasons. This has the advantage of being portable across non-UNIX operating systems,

but has the disadvantage of not allowing the use of standard UNIX utilities to manipulate the relations and databases

- support of backup operations by providing utilities for dumping entire large databases onto multiple tape volumes cleanly, using standard labels.

### 3.2 Hardware Decision

We will comment briefly on our hardware decision. We evaluated the following systems: DEC VAX 785 and 8600, CCI Power 6/32, Gould PN 6000 and PN 9000 and Pyramid 90x and 90Mx. We had several criteria and constraints:

- There was very limited floor space available,
- we needed Ingres in early 1985,
- to communicate over Ethernet with the IBM/MVS we were using an Auscom 8911A, so interim XNS support as well as TCP was necessary,
- a very clean UNIX port was important, as was the ability to run BSD and System V at the same time,
- reliability was critical,
- an end-user orientation and solid field support organization in our area were essential,
- we were very sensitive to price / performance.

We had experience with the CCI from a former project for the US Army. We were unable to get an 8600 running ULTRIX in the time frame needed and the 785 / 8600 were physically large and not exceptionally good at disk I/O. Gould did not yet run Ingres and we had concerns about disk I/O performance and stability of the port. We also discussed with Relational Technology their porting schedule for new releases, and learned that the Pyramid would be one of their first UNIX ports for each new release. Pyramid was a clear winner on all points and we already had several months experience using a 90x. We were able to obtain information about company plans, so we were aware of coming high throughput disk I/O controllers and faster processors. Benchmark results we ran were encouraging.

We are currently running four Pyramid 98x's, with a total of over 8 gigabytes of disk space. Two of the Pyramids are in production use, one is for development, and the fourth is currently used for benchmarks and kernel modification testing and will be phased over to a production status as more applications come up.

## 4. Distributed Computing Model

### 4.1 Model and Local Extensions

We adopted the client — server model of distributed computing. Here we describe this as related to distributed database integration; for a more complete description see [3]. We began using and porting the Sun RPC/XDR [5] as soon as it was distributed over Usenet in April 1985. We developed versions that worked over TCP, XNS and RS-232, and made modifications and fixes to the distributed code. We implemented RPC/XDR for integers and strings under IBM MVS/CICS in COBOL/Assembler so native processes could do RPCs across the Ethernet via the Auscom; we also developed an interface to the Auscom. We developed an ASCII version of RPC over RS-232 to a UNIX gateway server which we wrote on the Pyramid. This enables MUMPS systems to connect via serial lines to the gateway for network access to remote systems via RPC. The additional serialization / deserialization is done under UNIX in the intelligent gateway.

### 4.2 Example

These capabilities have been in production use for some time. For example, in December, 1985 we removed the VSAM dataset for patient identification from the IBM 3081 without advising users. Since then, COBOL applications on the IBM access the data in a large centralized Ingres database server on the Pyramid via RPC calls across the network. Similarly, we developed programmatic interfaces from MUMPS

enabling MUMPS applications to do RPC calls into this (and other) databases. We thus replaced a collection of several mutually inconsistent PID files in different systems (with measured inconsistencies of >33%) with a single system accessible to all. We have published an RPC Network Services Catalog enabling programmers in different environments to manipulate data in remote systems *without needing to know about the structure of the remote databases*.

#### 4.3 Server — ization

All of our Ingres — based applications are "server-ized". The front-end forms applications *do not contain explicit Equel/C calls to the shared databases*. Rather, data is accessed by issuing an RPC. All database access, thus all the Equel and knowledge of the schema, resides in the RPC server programs. These servers may be located in the same machine, or somewhere else out on the network. We have adopted this architecture for reasons of performance, resource conservation, and modularity. We have found that the Ingres back-ends are fast enough to support about a handful of our users. However, each back-end is huge; in our applications we see back-ends average about 1.0 Mbyte of non-shareable memory. (Of the 1.0 Mbyte of virtual address space about 0.3 Mbyte is resident if there are no repeat queries. Repeats would add about 100 to 150 Kbytes.) The shared text segment is about 0.5 Mbyte. Thus a typical 16 or 32 MB system can support only about 30 users (depending on working sets and considering memory consumption by front — end programs) before paging begins. Paging is the kiss of death for database transaction oriented applications. So, we needed a way to support several users, running common front-end functions, each sharing the back-ends with other users. To accomplish this we split applications into client and server pieces. The clients do forms manipulation and issue RPCs to the servers which may be on the local or a remote machine. The servers do the Equel/C accesses to the database. Multiple client applications can share the same server. This enables us to support upwards of five times the number of users than we could with the standard architecture.

We do have some problems with this approach. Since many of our RPCs sometimes return over 8K of data from the server, we need to use RPC using TCP, i.e., RPCs over dynamically created TCP connections, instead of using UDP. We have observed that the TCP opens and closes require too much overhead. (We have also determined that portmapper *does not* seem to be a bottleneck.) Use of permanent TCP connections between the clients and servers would solve this problem, but this would require (even more) major re-work of RPC/XDR. This approach does not solve the fundamental problem, which is that the back-ends cannot multi-thread requests. Therefore, users can see large variations in response times — this will happen when a long query is queued ahead of a short one waiting for the server. Another problem with this approach is that multiple RPCs cannot be bundled into a multi-statement update transaction in the client. This is because for each RPC the server will commit the update requested and there is no way for the client to back out the updates. Therefore, all transaction control must be in the server. However, a single RPC update can behave like an atomic transaction.

Schemes which attempt to load balance across common servers require rapid access to shared information or interprocess synchronization which are all too inefficient in 4BSD and in almost all implementations of System V shared memory (actually the Pyramid is an exception, since it supports shared memory and synchronization via hardware semaphores, obviating the need for system calls.) We note that most commercial operating systems support multi-threaded processes, or at least asynchronous disk I/O, so the real limitation is UNIX.

Server-ization also provides modularity benefits. Applications programmers can focus on the user interface and application flow, without concern for writing the database access code. Server programmers can focus on optimizing performance of the server, selecting indices and access methods, and on the structuring of multi-statement transactions, the control of locking and other technical and performance details. Furthermore, the separation of front-ends from back-ends makes possible the replacement of one component without impact on the other. It also makes possible access to multiple databases or DBMS's from a single front-end. Since most programming seems to be in the front-end, one could replace the back-end system entirely, transparently to the front-end programs and end-users. The interface is a well-defined, documented set of specifications. An additional advantage is that there is a uniform style of access to remote services, whether they be database management systems, number crunching services, expert system

servers, etc. The server-ization approach seems to work well for repetitive, parameterized transaction processing. Since RPC servers single-thread, this approach would not work well for ad-hoc queries which could tie up the service and block other clients.

#### 4.4 Heterogeneous Update Distribution

Like most distributed systems, we have had to deal with the problems of replicated data, and distributed updates in an environment where remote machines may not always be available. In analyzing our applications, we determined that in most cases, there is a "primary" site which should be updated immediately (and which may not be the machine the user is currently running on), and that the rest of the sites are "secondary", and do not have to be updated immediately, as long as the update is guaranteed to occur, and further, that it will be presented to each site in the correct sequence with the other updates for that site. We designed a *transaction server* which queues up transactions for each remote site, and delivers them in order. But if one or more sites is down, it does not stop the transaction server from delivering transactions to other sites, and resumes transmission to the down site when it comes back up. The transaction server also contains the intelligence to determine for each type of transaction that it receives, which remote sites need to receive the transaction. This frees each application from having to know all the other applications that have duplicates of the data, and makes adding a new application much easier. We considered the simplification offered by centralized control to be a good tradeoff against the complexity of distributed control for multi-site update propagation, however we have single-site failure. We address this by placing the transaction server on the same system that needs to be available as the database server for almost all current applications.

The transaction server does add some overhead to the processing of updates, so in special cases where one site is updating only one other site, we continue to use the Ethernet equivalent of a point to point link to allow direct communication.

### 5. Brief Application Descriptions

Rather than describe each application in detail, we will instead discuss the challenges that each application posed. Those interested in more detail on the actual applications should see the references [1, 4, 6, 7].

#### 5.1 Long Term Database (LTDB)

The LTDB is the central database of patients who have been seen at Johns Hopkins, and contains a summary of their medical history. Aside from the problems of system level interfacing and deciding on what a standard patient name looks like, the biggest problem that the LTDB posed was its size. The LTDB contains several relations each with over two million tuples plus secondary indices on these base relations. These relations generally consume more than 200 megabytes of disk space, each. Because of their size, it was not possible to use Ingres to sort these relations (which would have to be done on one physical disk) and so no primary index could be set up. Instead, we have had to use a base relation which is a heap, with numerous secondary indices to support the queries that are typically done. This has been fairly successful, although there is no clustering of the data, so we see a higher than necessary amount of disk seeking. We also see a higher than normal consumption of database locks since logically clustered data does not cluster physically onto the same pages. A secondary effect of this is unnecessary escalation to relation level locking. (Since Ingres defaults to level 3 consistency, rather than level 2 consistency, read locks are held within the transaction until escalation. See below for a discussion of how we address such problems.) We actually use a compressed heap, which saves considerable space and provides some performance advantage by denser record packing onto pages. We do not perform any storage operations that would cause a large penalty with the compressed heap organization.

Another problem is the non-uniform distribution of some of the data, particularly patient names. We have a combined index on the soundex for the patient's last name, the first four letters of the patient's first name, and the patient's date of birth. Normally, the first two parts (soundex and first name) are more than sufficient to limit the number of choices to under 200. However, there are about 100 names which exceed this number. In particular, if one looks up "Mary Smith" through this index, more than 43,000 matches will be returned. If the query is done for "Mary Smith", does not specify a birthdate, but does specify the

mother's maiden name (not a key field), the system will sequentially search all 43,000 matches to try to find a possible match. So even though the user specifies a reasonably restrictive search (patient's name, and mother's maiden name), the search still takes a very long time.

What we did to work around this problem was to add two relations to the database. The first contains each soundex code that occurs in the database, and the frequency with which it occurs. The second contains all occurrences of the first three letters of the first name (the first four letters was too large a set to be useful), along with the frequency. Now, before doing a search which may take forever, the application first looks up the frequency of the soundex and first name, multiplies them together, and compares this number against a threshold which represents approximately 200 names. If it is higher than the threshold, it prompts the user to enter more information (such as a first name, if one was not given, or a date of birth). If it is below the threshold, then the application proceeds with the retrieval. This is an example of an "application-level query optimization strategy" to avoid letting the DBMS optimizer select and implement a plan which would be operationally too expensive (and which would have severe impacts on concurrent updates.)

The other problem that the LTDB faces is that of concurrent access. Because much of the data is not clustered, retrievals (especially by name) tend to lock many more tuples than those which are being selected. This blocks other attempts to do updates, especially if the retrieving process locks a sufficient number of pages to escalate to a relation-level lock. In our applications, we have identified two types of retrievals, those that are part of a multi-statement transaction, and those that are not. The latter set does not really need as much protection from dirty reads. For this set, we have disabled the read lock protection, so that these retrievals do not block updating processes from accessing the data. The cost of this is a possible false retrieval, but with a patient population of over 2 million, it is very unlikely that two users would be accessing the same tuple at the same time. For retrievals in multi-statement transactions, where later updates will be dependent on the data retrieved, we still keep full read locking. In fact, we have the application acquire a write lock on retrieval if we are going to update that same tuple in order to reduce the possibility of deadlock and the resulting overhead for backout.

Application development with a large database also requires some additional caution. Typically, we develop with a database that is a small subset of the full LTDB. But we have to look carefully at the query plan for each transaction, to make sure that it will not try to search sequentially, or worse, to sort a 2 million tuple relation. Even after it has been carefully checked on the subset database, a final check must be done against the true database, since the query optimizer might choose a different plan when confronted with a very large relation.

In addition to access from UNIX systems, access to the LTDB is provided from programs running on the IBM MVS CICS system as well as to programs running on the several MUMPS systems. Common screens were developed across these different environments. Thus, one can walk up to a 3278 terminal, or an ASCII asynchronous terminal connected to any one of several computers and retrieve patient information from a common, shared database server system via essentially identical user interfaces. One might say that the mainframe acts as a terminal concentrator front-end to the Pyramid-based LTDB system for this application. Typical response times for simple retrievals are no longer than one second, regardless of which machine is running the front-end. However, complex retrievals returning multiple records can take 15 seconds. We have measured performance improvements of a factor of two to three between Ingres 3.0, no repeats, and Ingres 5.0 with repeats using our actual queries for the tests. We have also benchmarked Sybase (on a Sun 3/160) vs. Ingres using a large subset of the LTDB.

## 5.2 Clinical Appointing and Scheduling system (CAS)

The CAS system allows the outpatient clinics to schedule appointments for patients based on templates that each clinic can set up which describe when resources (e.g., doctors) are available. CAS also allows the entry of exceptions, such as vacations, holidays, or doctors trading slots for one day. Our initial design for the application had each clinic's templates stored in third normal form. Each time we needed the current schedule for a particular date, we would do a join which would instantiate the template for that date. Unfortunately, the instantiation required three transactions, one of which was a four table non-equijoin. In the development stages (before real data was introduced) this would take approximately one minute to

complete - far too long for production users.

A solution to this problem is "de-normalization". While we retained the original relations for storing the templates, we added a new relation, *sched* which represents the instantiation of all of the clinics' templates for the next 18 months. Now, instead of having to do the three transactions, and the non-equijoin, we instead do a simple retrieval against a single table to find out if a doctor has time to see a patient on a specified date.

There are two disadvantages to this solution. First, we had to set up a job which runs each night to chop off one end of this relation, and to instantiate the new day, 18 months from now. Second, the administrative programs to modify a clinic's templates became much more intricate. Now they must modify not only the relations where the templates are stored, but they must also go in and change *sched* so that it correctly reflects the new templates. To do this accurately, the administrative programs must have a firm picture of what the schedule looked like before the change, and what it looks like after the change. This is not always trivial, as in the case where a doctor's schedule changes on July 1st, so he has one set of templates that go through June 30th, and another that start on July 1st, and then the doctor decides to go on vacation for the last week of June and the first of July. Further, there is the problem of deleting a modification. If the doctor in the above example decides to cancel the vacation, the administrative screens must be able to "rebuild" the two weeks from the templates. This can cause problems during concurrent usage.

### 5.3 Radiology Information System

A new Radiology appointment scheduling and charge capture system has been developed under Ingres and has recently gone into production. In addition, a distributed film jacket tracking system, which produces bar code labels and contents labels has been in use for several months. The system runs on one of the Pyramids and uses by RPCs to the LTDB on a different Pyramid for patient identification. When PID information changes, which is not infrequent, new labels must be produced, so PID update transactions must propagate to the labeller system via the transaction server.

A system which keeps on-line radiology exam reports for one year has been in use for many months. For reasons too complex to be believed, reports are entered into the IBM from special workstations, and other reports are entered into the UNIX systems. RPCs take care of sending reports across to the "other" system, however the UNIX system is the actual on-line report archive and supports MIS functions for the department. Considerable work has gone into optimizing these systems. In addition to methods discussed above, static tables (dictionaries) are cached in applications on startup to reduce the join cardinality to a maximum of two whenever possible. Originally, the report archive was denormalized to the extent of having repeating fields to avoid joins; we have changed this to a cleaner design. The report system is server-ized. A report transcription system and report retrieval system have been developed as network services. We are making this application part of a larger clinical results database which will include laboratory, radiology and other ancillary reports.

### 5.4 Emergency Room (ER)

The ER system poses some special problems. The major one is the need for a high degree of availability. While the LTDB should be up 24 hours a day, 7 days a week, there are sufficient contingency plans in place for unexpected down time. There is enough of a paper trail that Medical Records can determine what history numbers have been assigned, etc. In the ER system, patients' medical information is captured into a database. This information includes their vitals signs, some test results, as well as the patients' description of the problem and what allergies and medications they may say they have. If the system goes down, this information must still be available for all patients currently being seen, regardless of where in the treatment process they are.

The solution we are developing hinges on a small personal computer which will be connected to the application machine by a dedicated RS-232 line. It will keep a record of all active patients in the ER. When something is written to the ER database, a similar record will be written to the personal computer. If the production system goes down, the personal computer will print out a report for each of the active patients. This report contains all of the information about the patient that has been collected up to this point. This mechanism will then allow treatment to continue, and still provide the doctors and nurses with

the information that has been collected under the ER system. When the production system is available, all data collected during the down time can be entered into the system.

### *5.5 Clinical Results Database (CRDB) and Clinic Workstation System (CWS)*

Two applications are being developed under Sybase. One of these, the clinical workstation system (CWS) will use 2 or 3 workstations on each of 50 nursing units to support inpatient clinical management. We are developing the pilot system for use on two nursing units by mid-1987; this will use diskless Sun-3 workstations (with a disk server for booting and paging) and will communicate with a Sybase server for the CWS database and a Clinical Results Database (CRDB) containing "recent" test results for patients who are or have been recent in- or out-patients. The CWS workstations will also communicate with IBM systems using both RPCs and 3270 emulation. CWS will directly support physicians and nurses. The pilot will use the SunView windowing system as the front-end. The CRDB will be used throughout the institution from terminals, workstations and PCs. Both CWS and CRDB need to support high volume on-line transaction processing with a concurrent mix of inserts, updates and retrievals. CWS will process all admit, transfer, discharge and change transactions originating from the IBM mainframe as RPCs (about 2,000 per day); eventually 50,000 to 100,000 transactions per day into CRDB will originate from automated instrumentation in patient rooms. The 150 workstations will also generate many retrievals and updates.

The CRDB will have over 100 active concurrent users fetching results by first identifying patients and then qualifying desired results. Thirty days of clinical lab data will be available, one year of radiology reports and pathology reports will be on-line; EKG and neurometrics reports are also planned. The Lab Medicine computers will generate records for appending; there will be about 75,000 records per peak five hours, and about 1,000 radiology reports added per peak five hours. Fast response will be essential since users are now accustomed to BDAM response times on a 3081.

## *6. Problems*

Our experiences have pointed out to us several areas that need improvement, both in database systems in general, and in features that UNIX needs to support to allow the database systems to work better.

### *6.1 Problems: DBMS*

The following problem areas seem to exist with most software based DBMS's:

- designed for functionality, but at the expense of performance
- performance restricted by the software architecture of the product
- reliance on the UNIX file system
- not designed to provide relational and business integrity constraints within the database, thus increasing application program complexity, size and maintenance problems; also causing many accesses from the application to check integrity
- not architected for high availability; not designed for efficient operational control
- inadequate provisions for maintenance needs and application configuration management

The common complaint is always that the database systems need to be faster. Below is a discussion of some ways this can be achieved. One major area of problems is that the processes associated with databases tend to be very large, and hence quickly fill up the memory of the machine, long before the CPU is completely utilized. Architectures which require large per-user database processes are doomed to failure. Our own machines succumb to massive amounts of paging long before the CPU or disk bandwidth (excluding paging) have been used up. A much more efficient architecture would be to have one server for all the processes on the system, but this server must be multi-threaded, and currently UNIX does not provide the hooks to make this possible (see the next section). Also note that in a single server architecture, locking and intelligent caching schemes become much easier to manage, since they are no longer distributed among numerous processes. In addition to being easier, they become less expensive: locking

can be done without all the overhead of system calls and caches can be shared among all users efficiently. Task switching can be managed by the DBMS process much faster than by the native operating system (UNIX). True compiled database (SQL) procedures, which are cached and which can be parameterized need to be provided. Control flow needs to be embedded into SQL for this to be really useful. Main memory should be mostly allocated to cache for indices, system catalogs, stored compiled procedures and frequently used data pages. Main memory should not be wasted by back-ends which replicate data for each process.

The other major area in need of development is support for production systems. To gain acceptance in the "real world", database systems must be prepared to run non-stop. This means that it must be possible to back up the database without shutting down access, to be able to modify schemas without affecting currently running applications, and to do whatever other software maintenance may be necessary without having to shut everyone out of the database (including updating optimizer statistics.) The real advantage of relational database management is the ability to make changes without massive redesign and re-implementation. In the real world, databases will always change. But the cost of dumping and re-loading databases can easily be *days* of downtime! Thus, the relational advantage can be fully realized only by enabling schema and other maintenance changes to occur concurrently with transaction processing. Since the natural unit of integrity is a database, rather than a relation, backups should dump entire databases and should disallow relation level dumps.

Also, much more elaborate facilities need to be provided to analyze performance in a production system. It should be possible to determine what user or process is holding locks (very useful for determining who has just relation-level locked some part of the database). Also, it should be possible to get statistical reports on transaction traffic to facilitate tuning the database, and also for billing the user.

As discussed above, centralized control of integrity within the DBMS itself, rather than enforcement by the front-end applications programs, is necessary to protect the data. To enable "business integrity rules" in addition to entity and referential integrity, triggers should be provided for use within the compiled procedures. Restrictions such as allowing only one database at a time to be open by an application are harmful; they cause aggregation of data that should be separated logically. This applies especially to "dictionary" data — the collection of facts about an enterprise's entities — which may be needed by all applications.

## 6.2 Problems: UNIX

It is not clear that a DBMS needs much from an underlying general purpose operating system. In fact, it is probably true that use of the operating system facilities can lead to serious performance penalties. DBMS designers need to work towards eliminating operating system overhead. Issues include:

- cache management
- paging and swapping
- task switching
- memory sharing across DBMS tasks
- file system overhead and relationship of an OS file to a database or to a relation; translation overhead; space reclamation after record deletes
- scheduling resources
- non-blocking I/O
- general support for multiple threads through a process

There are several areas where UNIX could improve, to provide the flexibility or low level features that a database system needs. The first of these is to allow a user process to do asynchronous disk I/O. While the disk caching and scheduling system is appropriate for a general purpose timesharing system, it is not necessarily efficient for a machine doing primarily database work. It would be better if the database server process could issue non-blocking disk I/O requests, and implement its own scheduling scheme. This would

also allow the server to multi-thread simultaneous transactions.

With the rush to support file system switches, a "standard" contiguous-file file system should be incorporated into UNIX. In this way, data logically grouped together would also be grouped physically on the disk, and would lower overall seek activity. This is already done by several of the database manufacturers, but they are all done as "modifications" and are not standard, nor do they support the use of other UNIX utilities. If a contiguous-file file system were implemented under a file system switch, then it should be possible to provide the benefits of the contiguous files, while still allowing other UNIX utilities to work in the new file system.

UNIX should provide a standard way for a logical file system to span more than one physical device. Several vendors are implementing disk striping for performance and this feature would also allow the use of files that are larger than one disk. Some machines support many of these features, as well as synchronization and access to shared memory without system calls, but DBMS vendors do not take advantage of available features, possibly because there is no standard implementation.

There should be a mechanism within UNIX to atomically support disk shadowing. This can be done at the application level, but the write to both disks is not atomic, and inconsistencies could develop. This feature becomes more important as UNIX moves into more traditional data processing areas, where reliability and some measure of fault tolerance are a necessity.

Shared library availability for memory and disk space conservation and run-time linking to enable all applications to use a new version of library modules at the same time are features needed by UNIX to address the demands of commercial applications. Screen manipulation libraries, whether for workstations or traditional terminals, require large amounts of code. Without widespread acceptance and use of shared libraries, each application duplicates the bulk of this code, wasting memory and causing more paging.

The file system itself has too much indirection and is too slow. UNIX was designed for a large number of small files; database systems usually have a small number of large files. Having to use system calls for access to shared resources, such as System V shared memory, or for synchronization, such as using a kernel-based lock driver, is enormous overhead for basic, simple operations.

At the utility level, UNIX needs better support for dumping files that span tape volumes and for maintaining a library of ANSI-labelled archived data.

## 7. Summary

Despite the issues and problems that we have encountered, OCS has developed a wide variety of applications in a relatively short period of time. We have a variety of applications on foreign machines (such as the IBM and MUMPS machines) going over the Ethernet to query a database running on a UNIX machine. In the two years that we have been heavily involved in these projects, we have seen a great deal of increased awareness, both among the database vendors and the hardware vendors, for the requirements that exist in a traditional data processing environment. We are seeing much more attention being focused on issues of reliability, recoverability, and the tools necessary to keep a production machine up and running. There is a danger, of course, of adding so much to UNIX that it becomes unmanageable, but we don't think that the requirements of a large production system and the UNIX philosophy are mutually exclusive. There are improvements that could be made. Hopefully these improvements will be based on industry standards, rather than being provided as vendor proprietary enhancements.

## References

- [1] S.G. Tolchin, "Overview of An Architectural Approach To The Development Of The Johns Hopkins Hospital Distributed Clinical Information System", *Journal of Medical Systems*, Vol. 10, No. 4, 1986, Page 321-338.

- [2] S.G. Tolchin, E.S. Bergan, et. al., "Transaction Processing Using Remote Procedure Calls (RPC) for a Heterogeneous Distributed Clinical Information System", *25th Annual Technical Symposium — Distributed Information Systems*, ACM and National Bureau of Standards, Gaithersburg, MD., June, 1986.
- [3] E.S. Bergan and S.G. Tolchin, "Using Remote Procedure Calls (RPC) for Supporting a Distributed Clinical Information System", *Proc. UniForum Conf.*, Anaheim, February, 1986.
- [4] S.N. Kahane, S.G. Tolchin, et. al., "Windows in the Hospital, or A Workstation-Based Inpatient Clinical System", *Proc. USENIX*, Denver, January, 1986.
- [5] Sun Microsystems Inc., "Remote Procedure Call Programming Guide", "External Data Representation Protocol Specification", "Remote Procedure Call Protocol Specification".
- [6] D. Schneider, et. al., "Capturing Clinical Information: An Automated Emergency Room System for the Johns Hopkins Hospital", *Hawaii Int'l. Conf. on System Sciences*, January, 1986.
- [7] D. Marquette and William Arrildt, "Radiology Film Tracking in a Distributed Clinical Network", *Proc. the Ninth Annual Symposium on Computer Applications in Medical Care*, November, 1985.
- [8] R. Lenhard, et. al., "The Johns Hopkins Oncology Clinical Information System", *Journal of Medical Systems*, Vol. 7, #2, 1983.
- [9] D. Bitton, D. J. DeWitt, and C. Turbyfil, "Benchmarking Database Systems: A Systematic Approach", Computer Science Department Technical Report #526, Computer Science Department, University of Wisconsin, December, 1983.

UNIX is a registered trademark of AT&T

PDP, VMS, ULTRIX, MicroVax and VAX are trademarks of Digital Equipment Corp.

INGRES is a trademark of Relational Technology Inc.

Ethernet and XNS are trademarks of Xerox Corp.

OSx, 90x, 90Mx and 98x are trademarks of Pyramid Technology Corp.

IDM and IDEL are trademarks of Britton-Lee, Inc.

Oracle is a trademark of Oracle, Inc.

Unify is a trademark of Unify, Inc.

Informix is a trademark of Informix, Inc.

M/VX is a trademark of InterSystems, Inc.

Sun 3 is a trademark of Sun Microsystems Inc.

Power 632 is a trademark of Computer Consoles, Inc.

PN 6000 and PN 9000 are trademarks of Gould Inc.

MVS, CICS, IMS, and IBM are trademarks of International Business Machine Corp.

DataServer and DataWorkbench are trademarks of Sybase, Inc.







